HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication

Donghang Lu
Purdue University

Thomas Yurek
University of Illinois at Urbana-Champaign

Samarth Kulshreshtha
University of Illinois at Urbana-Champaign

Rahul Govind
University of Illinois at Urbana-Champaign

Aniket Kate
Purdue University

Andrew Miller
University of Illinois at Urbana-Champaign

ABSTRACT

Multiparty computation as a service (MPSaaS) is a promising approach for building privacy-preserving communication systems. However, in this paper, we argue that existing MPC implementations are inadequate for this application as they do not address fairness, let alone robustness. Even a single malicious server can cause the protocol to abort while seeing the output for itself, which in the context of an anonymous communication service would create a vulnerability to censorship and de-anonymization attacks. To remedy this we propose a new MPC implementation, HoneyBadgerMPC, that combines a robust online phase with an optimistic offline phase that is efficient enough to run continuously alongside the online phase. We use HoneyBadgerMPC to develop an application case study, called AsynchroMix, that provides an anonymous broadcast functionality. AsynchroMix features a novel MPC program that trades off between computation and communication, allowing for low-latency message mixing in varying settings. In a cloud-based distributed benchmark with 100 nodes, we demonstrate mixing a batch of 512 messages in around 20 seconds and up to 4096 messages in around two minutes.

KEYWORDS

HoneyBadgerMPC; Robustness; Fairness; Asynchronous Mixing; Anonymous Communication

ACM Reference Format:
Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. HoneyBadgerMPC and AsynchroMix: Practical Asynchronous MPC and its Application to Anonymous Communication. In *2019 ACM SIGSAC Conference on Computer& Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3319535.3354238

1 INTRODUCTION

Millions of users employ the Tor [43] network to protect the anonymity of their communication over the Internet today. However, Tor can only provide a weak form of anonymity against traffic analysis [42] and has been successfully attacked using strong adversaries [14, 76]. Furthermore, emerging applications such as distributed ledgers (or blockchains), thanks to their close relation with payments and the financial world, demand a stronger form of anonymity [48, 52]. For example, even the use of zero-knowledge proofs in blockchains [13, 66, 75] is undermined unless users submit transactions through a Tor-like service. Designing and implementing practical and scalable systems for anonymous communication with stronger anonymity guarantees is, therefore, an active and important area of research and development [3, 33, 49, 59, 77].

Anonymous Communication from MPC. Secure multi-party computation (MPC) is a natural approach for building distributed applications with strong privacy guarantees. MPC has recently made great strides towards practical implementation and real-world deployment and consequently, several general-purpose compilers (or front-ends [51]) and implementations are now available supporting a range of performance and security tradeoffs [4, 8, 15, 26, 40, 55, 56, 78]. Recent implementation efforts [8, 26, 73] have bolstered their security guarantees by focusing on the malicious rather than semi-honest setting (i.e., they tolerate Byzantine faults), and can scale to larger networks (e.g., more than 100 servers) while tolerating an appreciable number of faults. Further, in contrast to early MPC realizations centered around one-off ceremonies [16, 17], there has been increased interest in the MPC system-as-a-service (MPSaaS) [3, 8, 46, 65] setting, where a network of servers continuously process encrypted inputs submitted by clients. As scalable and maliciously secure MPSaaS becomes increasingly practical, there's an increasingly more convincing argument that it can be successfully used for highly desirable internet services such as anonymous communication.

The Need for Robustness in MPC. Despite the aforementioned progress towards practical MPC, in this paper, we highlight robustness as an essential missing component. All of the MPC implementations we know of do not guarantee output delivery in the presence of even a single active fault. Even worse, these implementations do not guarantee fairness, in the sense that an adversary can see the output even if the honest servers do not. In the context of an anonymous communication service, unfair MPC could be catastrophic since an adversary could link the messages of clients who retry to send their message in a new or restarted instance. Thus the primary goal of our work is to fill this gap by advancing robustness in practical MPC implementations and demonstrating the result through a novel robust message mixing service.

CCS '19, November 11–15, 2019, London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6747-9/19/11...$15.00
https://doi.org/10.1145/3319535.3354238

Challenges in Providing Robust MPC. For MPC based on additive (n-of-n) secret sharing such as SPDZ [40] and EMP [78], the guaranteed output is inherently infeasible. However, even among guaranteed output protocols based on Shamir sharing, we find that the vast majority [10, 38, 39, 41, 53] are sensitive to assumptions about network synchrony. In short, their confidentiality and integrity guarantees rely on synchronous failure detectors, such that if a server is temporarily unresponsive, then it is "timed out" and ejected from the network and the fault tolerance among the surviving servers is reduced. If t honest parties are timed out, e.g., because of a temporary network partition, then a single corruption among the remaining servers could compromise the client's confidential inputs. Hence for a robust distributed service based on an MPC, we would desire safety properties even in an asynchronous network. In this setting, a Byzantine fault tolerance of $t < n/3$ is a lower bound even for agreement tasks that do not require any confidentiality.

Our Approach: Asynchronous MPSaaS. To address the above challenges, we base our message mixing service, AsynchroMix, on a new MPC implementation, called HoneyBadgerMPC, which is the first to guarantee fairness and output delivery in a malicious setting without depending on network timing assumptions. AsynchroMix proceeds in asynchronous epochs, wherein each epoch the system selects a subset of k clients and mixes their inputs together before publishing them. Unlike HyperMPC [8], which relies on a central coordinator service, HoneyBadgerMPC employs asynchronous broadcast protocols to receive secret shared inputs from untrusted clients and initiate mixing epochs in a robust and distributed way. Like many MPC protocols, HoneyBadgerMPC relies on the online/offline preprocessing paradigm. In our protocol the cost of the offline phase is comparable to that of the online phase, hence it can run continuously in the background as mixing proceeds. While the online phase is entirely robust, more efficient (but non-robust) protocols are chosen to generate preprocessing elements in the offline phase. In this way, less work is required overall and a buffer of preprocessed values can be used to guarantee robustness in the presence of faults.

Realizing Low-Latency, Robust Mixing. We evaluate two approaches for mixing inputs in MPC. The first is straightforward and implements a switching network [34] that requires $\log^2 k$ rounds and $O(nk \log^2 k)$ communication to shuffle k client inputs. To improve on this, we present PowerMixing, a novel mixing technique for reducing the number of rounds to two and the communication overhead to only $O(nk)$ by increasing computation to $O(nk + k^3)$ per node. We show that this allows for messages to be mixed with a lower latency than we could otherwise achieve, with larger mixes being available to servers with more computational power.

To summarize our contributions,

- **Robust MPC System-as-a-Service**. We advocate for a new operating point for MPC implementations, which features a robust online phase, but an efficient non-robust offline phase used to fill a buffer of preprocessing values. This fills a gap between protocols from the literature, which forego an important security property (asynchronous safety) in order to provide a robust offline phase, and implementations, which are not robust at all. We also show how to use fully-distributed asynchronous broadcast primitives, rather than a central cloud coordinator (like MATRIX [8]), to receive client inputs and initiate MPC computations.
- **Novel MPC program for mixing.** We design and implement a novel MPC program that can mix an arbitrarily large number of messages in only two communication rounds. We evaluate this program against a switching network implementation and show the operating points at which it demonstrates mixing with lower latency. We also demonstrate a method to create arbitrarily many powers of a shared secret in one online communication round, which may be of independent interest.
- **First implementation of robust asynchronous MPC.** As a practical contribution, our prototype offers the first implementation of asynchronous MPC primitives with the guaranteed output which may be employed for robust secure computations beyond anonymous broadcast. In our cloud-based distributed experiments, we show it is practical to mix inputs from up to $k = 4096$ clients using $n = 100$ servers located across five continents just in a few minutes of end-to-end latency. Additionally, using our novel low-latency mixing program, we can mix a more modest $k = 512$ messages in just over 20 seconds.

2 PRELIMINARIES: MPC BASED ON SHAMIR SECRET SHARING

Our standard MPC setting involves n parties $\{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$, where up to $t < n/3$ of those can be compromised by a Byzantine adversary. HoneyBadgerMPC relies on many standard components for MPC [10, 28, 31, 41] based on Shamir secret-sharing [71]. Here, we detail the most relevant techniques and notation.

2.1 Shamir Secret Sharing and Reconstruction

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

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

Figure 1 outlines the standard approach [10, 28, 30, 31] for robust decoding in this setting, Robust-Interpolate. First, we optimistically attempt to interpolate a degree-t polynomial ϕ after receiving any $t + 1$ shares. If the resulting ϕ coincides with the first $2t + 1$ shares received, then we know it is correct. If the optimistic case fails, we wait to receive more shares and as they arrive to attempt to correct errors. In the worst case, we receive t incorrect shares and need to wait for $3t + 1$ total shares before we can correct t errors and find a degree-t polynomial that coincides with all $2t + 1$ honest shares.

In Appendix A we discuss implementations of RSDecode and Interpolate. We use FFTs to achieve robust decoding with quasi-linear overhead (i.e., incurring an $O(n \log^2 n)$ computational cost), rather than superlinear algorithms based on Vandermonde matrix multiplication which incur $\approx O(n^2)$ overhead.

Algorithm Robust-Interpolate

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

Figure 1: Robust Polynomial Interpolation

Batch reconstruction. We recall an algorithm for the amortized batch public reconstruction (BatchRecPub) of t-sharings for the $t < n/3$ setting by Damgård and Nielsen [41] in Figure 2. The idea is to apply a Vandermonde matrix M to expand the shared secrets $[\![x_1]\!], \dots, [\![x_{t+1}]\!]$ into a set of sharings $[\![y_1]\!], ..., [\![y_n]\!]$. In the first round, each server $\mathcal{P}_j$ locally computes their shares of each $[\![y_i]\!]^{(j)}$ and sends it to $\mathcal{P}_i$. Each $\mathcal{P}_j$ then uses Robust-Interpolate to reconstruct a different share y_j. In the second round, the servers exchange each y_j, and again use Robust-Interpolate to recover $x_1, ..., x_{t+1}$. When defining an MPC program, we use the notation $x_i \leftarrow \text{Open}([\![x_i]\!])$ for reconstructing an individual share, implicitly making amortized use of the BatchRecPub protocol.

2.2 SSS-Based MPC

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

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

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

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

Offline phase. In order to fulfill the computational needs of our online phase, we need to generate a steady supply of Beaver Triples offline (prior to when inputs for an MPC circuit are given). As the offline phase can be run for an indefinite amount of time, we relax the robustness requirements and focus on more efficient protocols. In this way, the offline phase can proceed with less work while still gradually building up a buffer and allowing for guaranteed output in the online phase.

Protocol BatchRecPub

- Input: $[\![x_1]\!], \dots, [\![x_{t+1}]\!]$
- Output: $x_1, \dots, x_{t+1}$
- Procedure (as server $\mathcal{P}_i$):
  1. Let M be the $(n, t + 1)$ Vandermonde matrix $M_{i,j} = \alpha_i^j$ evaluating a degree-t polynomial at $(\alpha_1, ..., \alpha_n)$.
  2. Compute $([\![y_1]\!], \dots, [\![y_n]\!])^T := M([\![x_1]\!], ..., [\![x_{t+1}]\!])^T$
  3. *(Round 1)* For each j, send $[\![y_j]\!]$ to party $\mathcal{P}_j$.
  4. Wait to receive between $2t + 1$ and n shares of $[\![y_i]\!]$ and decode y_i using Robust-Interpolate.
  5. *(Round 2)* Send y_i to each party P_j.
  6. Wait to receive between $2t+1$ and n values y'_j, then robustly decode $x_1, ..., x_{t+1}$ using Robust-Interpolate.

Figure 2: Batch Reconstruction [10, 28, 41]

The first step of the offline phase is randomness extraction [10], where secret-shared random values are produced from the contributions of different servers. To produce t-sharings of random elements of $\mathbb{F}_p$, we apply an (n, n) hyperinvertible matrix M, (concretely, a Vandermonde matrix) and compute

$$([\![r_1]\!], ..., [\![r_n]\!]) := M([\![s_1]\!], ..., [\![s_n]\!])$$

where each $[\![s_i]\!]$ is contributed by a distinct server $\mathcal{P}_i$, and we output $[\![r_1]\!], \dots, [\![r_{t+1}]\!]$. The choice of M ensures the $[\![r_i]\!]$ are random and unknown, despite of the influence of t corrupt parties. To check that the secret sharings are of the correct degree, $2t + 1$ of the servers attempt to reconstruct one column each of $[\![r_{n-2t-1}]\!], \dots, [\![r_n]\!]$. The hyperinvertibility property of M ensures that if all of the inputs are of the correct degree, then so are all of $[\![r_1]\!], \dots, [\![r_{t+1}]\!]$. Since all n parties must be online to provide input for this process, this cannot guarantee output if any parties crash.

To generate Beaver triples, we make use of random double sharings, which are t- and $2t$-sharings of the same random value $[\![r]\!]_t$ and $[\![r]\!]_{2t}$. For this we use RanDouSha [10, 41], wherein each server contributes a pair of shares, $[\![s_i]\!]_t$ and $[\![s_i]\!]_{2t}$. The first $t + 1$ pairs $[\![r_1]\!]_{\{t,2t\}}, \dots, [\![r_{t+1}]\!]_{\{t,2t\}}$ after applying M are taken as output, and the remaining $2t + 1$ pairs are reconstructed as a checksum (by one server each). All together, this protocol is given in Figure 3.

Given the double sharing, we generate a Beaver triple by generating random shares $[\![a]\!]_t, [\![b]\!]_t$, calculating $[\![ab]\!]_{2t} = [\![a]\!]_t \cdot [\![b]\!]_t$, and performing degree reduction:

$$[\![ab]\!]_t := \text{Open}([\![ab]\!]_{2t} - [\![r]\!]_{2t}) + [\![r]\!]_t.$$

Besides random field elements and multiplication triples, the offline phase is also used to prepare random bits, and k powers of random elements using standard techniques [36]. In general, we can implement any necessary preprocessing task by combining the above two ingredients. The overall cost of the offline phase is summarized

Protocol RanDouSha

- Input: pairs $\{[\![s_i]\!]_t, [\![s_i]\!]_{2t}\}$ contributed by each server
- Output: $[\![r_1]\!]_t, [\![r_1]\!]_{2t}, \dots, [\![r_{t+1}]\!]_t, [\![r_{t+1}]\!]_{2t}$
- Procedure (as server $\mathcal{P}_i$):
  (1) $[\![r_1, \dots, r_n]\!]_t \leftarrow M([\![s_1]\!]_t, \dots, [\![s_n]\!]_t)$
  (2) $[\![r_1, \dots, r_n]\!]_{2t} \leftarrow M([\![s_1]\!]_{2t}, \dots, [\![s_n]\!]_{2t})$
  (3) Each party $\mathcal{P}_i$ where $t + 1 < i \leq n$ privately reconstructs $[\![r_i]\!]_t$, $[\![r_i]\!]_{2t}$ and checks that both shares are of the correct degree, and that their 0-evaluation is the same. Reliable-Broadcast `OK` if the verification succeeds, `ABORT` otherwise.
  (4) Wait to receive each broadcast and abort unless all are `OK`
  (5) Output $[\![r_1]\!]_t, [\![r_1]\!]_{2t}, \dots, [\![r_{t+1}]\!]_t, [\![r_{t+1}]\!]_{2t}$

Figure 3: Generating random double sharings [10, 36, 41]

by the number of batch reconstructions and the number of random shares needed. We summarize the offline costs for our two mixing approaches in Section 5.

2.3 Asynchronous Reliable Broadcast and Common Subset

We employ an asynchronous reliable broadcast primitive in order to receive client inputs. A reliable broadcast (RBC) protocol satisfies the following properties:

- *(Validity)* If the sender (i.e., the client in our case) is correct and inputs v, then all correct nodes deliver v
- *(Agreement)* If any two correct servers deliver v and v', then $v = v'$.
- *(Totality)* If any correct node delivers v, then all correct nodes deliver v.

While Bracha's [20] classic reliable broadcast protocol requires $O(n^2|v|)$ bits of total communication in order to broadcast a message of size $|v|$, Cachin and Tessaro [24] observed that Merkle trees and erasure coding can reduce this cost to merely $O(n|v| + n^2 \log n)$ (assuming constant size hashes), even in the worst case. The non-linear factor of this cost comes from the need to send branches of a Merkle tree created over the erasure-coded shares to ensure data integrity.

In order to reach an agreement on which instances of RBC have terminated, and to initiate each mixing epoch, we rely on an asynchronous common subset protocol [12, 23, 67]. In CommonSubset, each server begins with an input b_i (in our application each b_i is a κ-bit vector). The protocol outputs an agreed-upon vector of n values that includes the inputs of at least $n - 2t$ correct parties, as well as up to t default values. CommonSubset satisfies following properties:

- *(Validity)* If a correct server outputs a vector b', then $b'_i = b_i$ for at least $n - 2t$ correct servers;
- *(Agreement)* If a correct server outputs b', then every server outputs b';
- *(Totality)* All correct servers eventually produce output.

To stick to purely asynchronous primitives, we concretely instantiate CommonSubset with the protocol from HoneyBadgerBFT [12, 67]; as an alternative, BEAT0 [44] is similar but offers more efficient cryptographic primitives. For small messages, the overhead for either protocol grows with n^2, although for very large messages it achieves linear overhead. If asynchronous liveness is not needed, then any partially synchronous consensus protocol, such as PBFT [25], would suffice here as well.

3 ROBUSTNESS IN MPC PROTOCOLS AND IMPLEMENTATIONS

In practice, distributed computing protocols should successfully protect against not just benign failures like system churn, but also network partitions and denial of service attacks. Distributed consensus protocols and systems employed in practice (e.g., [25, 54, 61]) put significant emphasis on achieving this robustness property, and the same also holds for prominent blockchain systems [5, 21]. Various notions of robustness have also been explored in the context of MPC, although we observe that the practical MPC tool-kits [4, 8, 36, 40] available today have not made a similar effort to incorporate this robustness. We therefore place a strong emphasis on achieving robustness in this paper.

In this section we evaluate the robustness of existing MPC implementations and protocols (summarized in Table 1), and use this evaluation to inform the design of HoneyBadgerMPC and AsynchroMix. We focus mainly on three forms of robustness: *fairness*, *guaranteed output*, and *safety in asynchronous communication setting*. In our work we focus on the MPC-System-as-a-Service model [3, 8, 46, 65], where clients submit secret inputs to servers for processing. However, in the usual MPC setting, the servers themselves are the clients. Thus for the sake of comparison, in this section we assume $n = k$ (where n is the number of servers and k is the number of clients). In this evaluation we leave implicit the need to agree on which inputs to include. In a synchronous network, MPC typically ensures that every honest party's inputs are included [11], while in an asynchronous network it is inherent that up to t honest parties may be left out [28]; to accommodate asynchronous protocols we assume the weaker definition. We also elide discussion of protocols and implementations that offer only semi-honest security, such as PICCO [80] or Fairplay [64], or that rely on trusted hardware [27].

Fairness and Guaranteed Output. *Fairness* is widely studied in MPC. Roughly speaking, it means that either all parties receive their output, or else none of them do [50]. Unfair protocols allow the adversary to peek at the output of the computation, while the honest parties observe the protocol fail. In the context of anonymous communications, unfair protocols pose a severe hazard of intersection attacks. For example, if a client retries to send their message in a new session with a different anonymity set, the adversary would learn which messages were common to both sessions [70]. To the best of our knowledge, none of the practical implementations of MPC aim to provide fairness against an active adversary. Instead, they focus on the weaker notion of *security with abort*, meaning that the honest parties reach consensus on whether or not the protocol aborts, which admits the intersection attack above.

Guaranteed output delivery is usually considered synonymous with robustness in MPC. It is a stronger notion than fairness that further requires that corrupt parties cannot prevent honest parties from receiving output. MPC Protocols based on n-of-n sharing

Table 1: Summary of Robustness in Active Secure MPC Protocols and Toolkits

	$t <$	Fairness	Guaranteed Output		Asynchronous		Complexity	Communication
Protocol Designs			Online	Offline	Safe	Live	Assumption	Overhead
BGW [6, 11]	$n/3$	●	●		○	○		quadratic
HN06 [53]	$n/2$	●	●		○	○	SHE	linear
BH08 [10],DN07 [41]	$n/3$	●	●	●	○	○		linear
DN07 [41]	$n/2$	●	●	●	○	○	Dlog	linear
DIK+08 [38, 39][1]	$n/8$	●	●	●	○	○		linear
COPS15 [29]	$n/2$	●	●	●	●	○	HE	quadratic
CHP13[28],CP17[31]	$n/4$	●	●	●	●	●		linear
CP15 [30]	$n/3$	●	●	●	●	●	SHE	linear
MPC Toolkits								
Viff [36]	$n/3$	○	○	○	●	○		quadratic
SPDZ [40, 55, 56]	n	○	○	○	●	○	SHE or OT	linear
EMP [78]	n	○	○	○	●	○	OT	quadratic
SCALE-MAMBA [4]	$n/2$	○	○	○	●	○		quadratic
HyperMPC [8]	$n/3$	○	○	○	●	○		linear
CGH+18 [26]	$n/2$	○	○	○	●	○		linear
This paper								
hbMPC	$n/3$	●	●	○	●	●		linear

for the dishonest majority setting $t < n$, such as EMP [78] as well as SPDZ [40] and its descendants, are inherently unable to provide guaranteed output. However, as long as $t < n/3$, then the online phase techniques for degree-t SSS described in Section 2.1-2.2 suffice. HyperMPC [8], for example, cannot guarantee output in the $t < n/3$ setting as it works with $2t$-sharings in the online phase. Unlike fairness, guaranteed output is primarily a concern for liveness rather than safety. A fair protocol that aborts can in principle be restarted with a new set of parties. In any case, the protocols we evaluate satisfy both or neither.

Asynchronous Safety and Liveness. MPC protocols that guarantee output typically fall into one of two camps. The first camp is based on (bounded) synchronous broadcast primitives and involves restarting the computation after detecting and eliminating one or more faulty parties. Such protocols can be unconditionally secure when $t < n/3$ [6, 10, 11, 41] and using cryptography can reach $t < n/2$ [41, 53]. Dispute resolution is also used by virtualized protocols that boost a low-resilience outer protocol (i.e., $t < n/8$) to $t < n/2 - \epsilon$ [38, 39].[2] However, we observe that these protocols rely on the ability to time out nodes that appear to be unresponsive, restarting the computation with the remaining parties. If t honest nodes are temporarily partitioned from the network, then *any* failures among the remaining parties could compromise the safety properties, including confidentiality. Using this approach to guarantee output, therefore, leads to an inherent trade-off between the liveness and safety properties—the more faults tolerated for liveness, the fewer tolerated for safety. Furthermore, the preference for performance would be to set the timeout parameter low enough to tolerate benign crashes, though this means even shorter duration network partitions weaken the security threshold among the remaining nodes.

We say a protocol has *asynchronous safety* if its safety properties hold even in an asynchronous network and up to t parties are corrupt.[3] The second camp of guaranteed MPC protocols relies on asynchronous primitives rather than dispute resolution, and proceed with the fastest $n - t$ nodes regardless of the network time [28–31]. We notice that since the MPC implementations do not aim for guaranteed output anyway and block on all n parties before proceeding, trivially satisfy this property.

Purely asynchronous MPC protocols [28, 30, 31] further guarantee liveness as well as safety without assuming bounded synchrony and broadcast channels. In this setting, even a replicated state machine task — without any secrecy properties at all — requires $t < n/3$, hence this is also a lower bound for asynchronous MPC. We know of two unconditionally secure asynchronous MPC protocols with linear overhead for the $t < n/4$ setting [28, 31], as well as a protocol for the $t < n/3$ relying on Somewhat Homomorphic Encryption (SHE) [30]. Other related protocols for asynchronous MPC include a constant-round online phase, independent of the circuit depth [32, 37]; however, these incur quadratic communication overhead in n.

Communication Overhead. Communication overhead is a critical factor in how well the network size n can scale. We mainly focus on amortized overhead over suitably large batches of operations. An MPC protocol has linear communication overhead if, for a given task, as a function of a network size n, the total communication cost grows with $O(n)$. In particular, this means that as additional nodes are added, the bandwidth required by each node remains constant. Besides communication overhead, we also discuss computation overhead in Section 6.1.

Informing the design of HoneyBadgerMPC. Concerns of intersection attacks are the primary reason not to use existing (unfair)

[2]We only consider the outer protocols of DIK+08,DIK10. By composing with an inner protocol, these can obtain security of $t = n/2 + \epsilon$, though this requires large randomly selected committees, and in any case, inherits the robustness and practicality of the inner protocol.

[3]Asynchronous safety is a requirement even for the stronger *partially synchronous* network model [45], where a protocol must guarantee safety at all times, but liveness only during periods of synchrony.

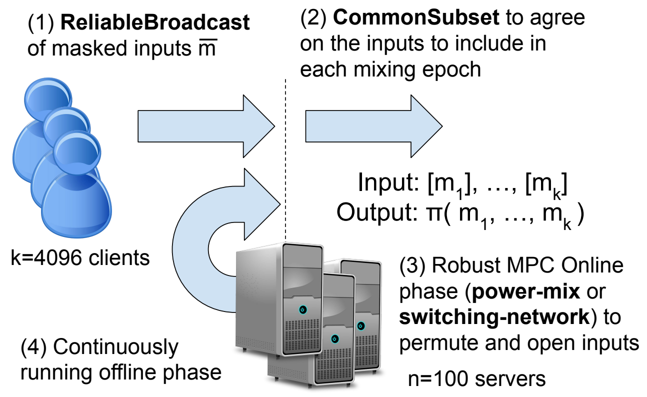
(1) **ReliableBroadcast** of masked inputs $\overline{m}$
(2) **CommonSubset** to agree on the inputs to include in each mixing epoch
Input: $[m_1], \ldots, [m_k]$
Output: $\pi(m_1, \ldots, m_k)$
k=4096 clients
(3) Robust MPC Online phase (**power-mix** or **switching-network**) to permute and open inputs
(4) Continuously running offline phase
n=100 servers

Figure 4: Overview of the AsynchroMix protocol

MPC implementations for AsynchroMix. We note that several recent works use a blockchain cryptocurrency and security deposits to provide financial compensation in case the protocol aborts unfairly [57, 58], though we aim to prevent such failures at all. We wish to avoid the tradeoff between safety and availability associated with asynchronous-unsafe protocols, which rules out protocols based on the synchronous broadcast.

This leaves the (partially) asynchronous protocols [28–31] as candidates. These guarantee liveness in the offline phase as well as the online phase, which means that service can continue indefinitely even if some nodes fail. However, these require either additional cryptography overhead or else offer less resilience ($t < n/4$ rather than $t < n/3$). To avoid these problems, our approach is to start from the unconditionally secure protocols for $t < n/3$ [10, 28], but relax guaranteed output in the offline phase. We envision optimistically running the offline phase ahead of time to build up a sufficiently large reserve of preprocessed values.

4 OVERVIEW OF ASYNCHROMIX AND HONEYBADGERMPC

AsynchroMix is an application of the MPC-System-as-a-Service (MPSaaS) [8] approach to the problem of anonymous broadcast communication. We consider a typical client-server setting for anonymous communication networks [43, 59, 77], where clients send their confidential messages to server nodes and server nodes mix clients messages before making them public. As our primary focus is robustness, we model an asynchronous communication network such that we must not make use of timeouts and do not rely on time-bound parameters to be correctly configured. The communication network is assumed to be under the adversary's control such that the adversary may arbitrarily delay messages, duplicate them, or deliver them out of order. For system liveness, we assume that the adversary cannot drop messages between two honest parties.[4]

[4]Although it is tempting to treat the network to be bounded-synchronous (bounded message delivery delays) [33, 70] and develop similar protocols using well-known message delivery time bounds and system run-time assumptions, deciding these time bounds correctly is a difficult problem to solve and will require frequent readjustments. Moreover, asynchronous protocol executions may often be faster than the protocol executions with the bounded-synchrony assumption as in most cases messages delivery may take significantly less time than timeout values.

As mentioned in Section 5, the goals of AsynchroMix include Safety (anonymity properties) as well as liveness - the system continues to work. The strong threat model includes a fraction being maliciously corrupted and does not rely on timing assumptions.

System Model: Assume a set of clients $\mathcal{C} = \{C_j\}_{j=1\ldots k_{pop}}$ with input messages m_j, who communicate to a set of n servers, $\{\mathcal{P}_i\}_{i=1\ldots n}$. We assume that at most $t < n/3$ of the servers are Byzantine corrupted by a global adversary, and similarly, any number of clients are corrupted as well. All servers are connected to each other over asynchronous channels, and every client is connected to all servers over asynchronous channels. The messages themselves are fixed sizes of $|m|$ bits (or field elements, depending on context).

AsynchroMix proceeds in sequential mixing epochs, where in each epoch we mix input messages provided by $k \leq k_{pop}$ clients. Fig. 4 offers a high-level overview of the process. The protocol satisfies the following security properties:

- **Anonymity (Safety):** During every mixing epoch, even when all but 2 selected clients are compromised, the adversary cannot link an included message m_j to its honest client C_j except with probability negligibly better than 1/2.
  Specifically, for input vector $m_1, \ldots, m_k$ from k clients, the output is a permutation $\pi(m_1, \ldots, m_k)$ such that the output permutation is at least almost independent of the input permutation.
- **Availability (Liveness):** Every honest client's input is eventually included in a mixing epoch, and every mixing epoch eventually terminates.

AsynchroMix is built upon a new MPC prototype, called HoneyBadgerMPC, which realizes secure computation through the use of asynchronous and maliciously-secure primitives. In particular, HoneyBadgerMPC makes use of asynchronous reliable broadcast to receive secret shared inputs from untrusted clients, and asynchronous common subset to reach agreement on the subset of clients whose inputs are ready and should be mixed in the next epoch. Each mixing epoch involves a standard robust MPC online phase based on Beaver triples and batched public reconstruction [10]. The offline phase [8, 10] runs continuously to replenish a buffer of preprocessing elements used by the online phase. The offline phase is optimistic in the sense that all server nodes must be online and functioning to replenish the buffer. These components are described in more detail below and illustrated overall in Figure 4.

4.1 Receiving Client Inputs using Preprocessing and Asynchronous Broadcast

Since clients are untrusted, we need a way to receive secret shared inputs while guaranteeing that the inputs are valid, consistent, and available at every server node. In principle, we could use Asynchronous Verifiable Secret Sharing (AVSS) [7, 22], though this would lead to additional communication and computation overhead. Instead, we make use of a preprocessing approach due to Choudhury et al. [29]. The idea is that for each input m from client C, we consume a preprocessed random share $[\![r]\!]$, which was generated in the offline phase and privately reconstructed to C (i.e., each server node sends their share of $[\![r]\!]$ to C, who robustly interpolates r). The client then blinds its message $\overline{m} := m + r$ and broadcasts the blinded message $\overline{m}$ ((1) in Figure 4). The servers then each locally compute their share $\llbracket m \rrbracket := \overline{m} - \llbracket r \rrbracket$, without leaking any information about m.

To broadcast $\overline{m}$, we make use of the asynchronous broadcast protocol ReliableBroadcast, which guarantees, roughly, that if any server receives m, then every correct server also receives m. More details on the reliable broadcast protocol are given in the Appendix.

4.2 Asynchronous Mixing Epochs

Each mixing epoch begins when servers have received inputs from enough clients. Servers must reach an agreement on a subset of k client inputs [2, 44, 67] which are deemed to be available for processing. Every epoch, this agreement is made using the asynchronous broadcast primitive CommonSubset [12]. At the beginning of CommonSubset, each server inputs its view of which client inputs are available for mixing. For honest servers, this will be the set of inputs for which a value has been received by ReliableBroadcast. The output of CommonSubset will be a set of k available inputs that will be used in the next mixing epoch.

4.3 Robust Online Phase

Once the inputs to a mixing epoch are determined, the mixing proceeds as an online phase of MPC, running one of two programs, power-mix or iterated-butterfly, as we detail in the next Section. The online phase itself is standard, based on Beaver triples [9], and only requires batch reconstruction of t-sharings, which in the $t < n/3$ setting we can achieve through Reed Solomon decoding [10, 41]. In Appendix A we discuss implementation improvements based on FFT.

4.4 Continuously Running Offline Phase

Since AsynchroMix is a continuously running service, the offline phase could be run concurrently to replenish a buffer of preprocessing values. Here latency is not critical, although it should ideally be efficient enough to keep up with the demand from the online phase. Our offline phase is an implementation of [10], the same as used in HyperMPC. It is based on decoding $2t$-sharings and therefore makes progress only when all n nodes are responsive. As mentioned earlier in Section 3, we consider it reasonable to use a non-robust protocol for the offline phase which runs ahead of time in order to provide a reserve buffer of preprocessed values. If one or more nodes fail, eventually the reserve will be depleted and clients will have to move to a new instance of the service.

4.5 Security Analysis of AsynchroMix

Theorem 4.1. *Assuming that sufficient preprocessing elements are available from a previously-completed offline phase, then the AsynchroMix protocol defined in Figure 5 satisfies the anonymity and availability properties defined earlier.*

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

Protocol AsynchroMix

```
• Input: Each client $C_j$ receives an input $m_j$
• Output: In each epoch a subset of client inputs $m_1, \ldots, m_k$ are
  selected, and a permutation $\pi(m_1, \ldots, m_k)$ is published where
  $\pi$ does not depend on the input permutation
• Preprocessing:
  – For each $m_j$, a random $\llbracket r_j \rrbracket$, where each client has received
    $r_j$
  – Preprocessing for PowerMix and/or Switching-Network
• Protocol (for client $C_j$):
  (1) Set $\overline{m}_j := m_j + r_j$
  (2) ReliableBroadcast $\overline{m}_j$
  (3) Wait until $m_j$ appears in the output of a mixing epoch
• Protocol (for server $\mathcal{P}_i$):
  - Initialize for each client $C_j$
      $\mathsf{input}_j := 0$      // No. of inputs received from $C_j$
      $\mathsf{done}_j := 0$       // No. of messages mixed for $C_j$
  - On receiving $\overline{m}_j$ output from ReliableBroadcast client $C_j$ at
    any time, set $\mathsf{input}_j := \mathsf{input}_j + 1$
  - Proceed in consecutive mixing epochs $e$:
  Input Collection Phase
      Let $b_i$ be a $|\mathcal{C}|$-bit vector where $b_{i,j} = 1$ if $\mathsf{input}_j > \mathsf{done}_j$
      Pass $b_i$ as input to an instance of CommonSubset
      Wait to receive $b$ from CommonSubset, where $b$ is an $n \times |\mathcal{C}|$
      matrix, each row of $b$ corresponds to the input from one
      server, and at least $n - t$ of the rows are non-default. Let
      $b_{\cdot,j}$ denote the column corresponding to client $C_j$.
      For each $C_j$,
          $\llbracket m_j \rrbracket := \begin{cases} \overline{m}_j - \llbracket r_j \rrbracket & \sum b_{\cdot,j} \geq t+1 \\ 0 & \text{otherwise} \end{cases}$
  Online Phase
      // Switch Network Option
        Run the MPC Program switching-network on $\{\llbracket m_{j,k_j} \rrbracket\}$,
        resulting in $\pi(m_1, \ldots, m_k)$
        Requires $k$ rounds,
      // Powermix Option
        Run the MPC Program power-mix on $\{\llbracket m_{j,k_j} \rrbracket\}$, resulting
        in $\pi(m_1, \ldots, m_k)$
      Set $\mathsf{done}_j := \mathsf{done}_j + 1$ for each client $C_j$ whose input was
      mixed this epoch
```

Figure 5: Protocol for asynchronous mixing of values.

The Switching-Network induces a random permutation, which is sampled from a nearly uniform distribution.

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

4.6 Comparing AsynchroMix with Other Strong Anonymity Solutions

We observe that most anonymous communication systems do not focus on robustness and thus cannot achieve strong availability guarantees in the presence of faults. For example, in protocols following mix-nets strategies such as [59, 60, 62, 69, 77], nodes encrypt/decrypt layers of encryptions of user/cover traffic or re-encrypt batches of messages, and many failures has to result in users resending their messages. Similarly, in protocols following DC-net strategies such as [33, 70], nodes collaborate to randomly permute a set of messages while decrypting those, and any participating node may abort the execution and force re-execution. In order for these protocols to handle failures, it is necessary to rely on synchronous network assumptions to timeout a node, potentially restarting a computation or requiring users to resend messages. This introduces many potential issues. The first is that compromised nodes may attempt to degrade performance, such as by stalling until the last moment before being timed out. Attempting to optimize the protocol for speed by reducing the timeouts would only make it more likely that honest participants who experience a fault would be removed, thus degrading security. More importantly, by DoSing some honest nodes during re-running, it is also possible to launch inference attacks leading to deanonymization [18, 70, 79]. On the other hand, most of these schemes can indeed maintain anonymity/privacy against much larger collusion among the nodes, while liveness requirements of AsynchroMix in the asynchronous setting mandate us to restrict the adversarial collusions to $t < n/3$ nodes.

Our approach to MPC mixing is closely related to MCMix [3], which implements an anonymous messaging system based on MPC. Instead of a switching network, they associate each message with a random tag and obliviously sort the tags using MPC comparison operations.

5 MPC PROGRAMS FOR MESSAGE MIXING

Once the inputs are selected, $[\![m_1]\!], \dots, [\![m_k]\!]$, each asynchronous mixing epoch consists of an online MPC phase, computing either the Iterated Switching Network or PowerMix MPC programs.

The first approach is based on an iterated butterfly switching network [34] which yields an almost-ideal random permutation of inputs. Each switch uses a secret-shared random bit from the offline phase and a single MPC multiplication. Overall this method requires $O(\log^2 k)$ asynchronous rounds. The communication and computation cost per server are both $O(n \log^2 k)$ per input.

As an alternative to the switching network, we present a constant-round protocol called PowerMix, based on Newton's sums. To mix a batch of k messages $[\![m_1]\!]$ through $[\![m_k]\!]$, the servers first compute the powers $[\![m_i^j]\!]$ where i, j range from 1 to k. We then locally compute the sums of each power, $[\![S_i]\!] = \sum_{j=1}^{k} [\![m_j^i]\!]$ and publicly reconstruct each S_i. Finally, we use a solver for the set of m_i using Newton sum methods. Ordinarily, computing $[\![m_i^j]\!]$ using Beaver multiplication would require at least $O(\log k)$ rounds of communication. However, in PowerMix we use a novel way to trade-off communication for computation, generating all the powers in a single round of communication by using some precomputed powers of the form $[\![r]\!]$,$[\![r^2]\!]$,. . .,$[\![r^k]\!]$. As a result, PowerMix only requires two rounds of communication to finish mixing.

MPC Program switch

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

$[\![c]\!] := [\![b]\!] \cdot ([\![i_1]\!] - [\![i_2]\!])$
$[\![o_1]\!] := 2^{-1}([\![i_1]\!] + [\![i_2]\!] - [\![c]\!])$
$[\![o_2]\!] := 2^{-1}([\![i_1]\!] + [\![i_2]\!] + [\![c]\!])$

MPC Program switching-network

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

Figure 6: Permutation based on a switching network

5.1 Option I: Switching Network

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

Switching networks are implemented in layers, where each layer applies a permutation to the inputs by conditionally swapping each pair. However, the resulting permutations are biased [1, 68]. For example, while a random Benes network can express every possible permutation, some permutations are more likely than others. Czumaj and Vöcking showed that $O(\log k)$ iterations of random butterfly networks (each of which consists of $O(\log k)$ layers) provide adequate shuffling [34] in the sense that the combined permutation is nearly uniform. The round complexity of the switching network is $O(\log^2 k)$, and the overall communication cost is $O(k \log^2 kn)$ considering there are $O(\log^2 k)$ layers in total and $O(k)$ multiplications are needed in each layer. Computation cost is $O(k \log^2 kn)$ since $O(k \log^2 kn)$ multiplications are needed in total. (See Figure 6 for a secure switching network instantiation with standard MPC operations.)

Table 2: Summary of Online Phase computation and communication cost overhead (per client input) for Iterated Butterfly and PowerMix MPC programs

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

5.2 Option II: PowerMix

To contrast with the switching network, we propose a novel protocol PowerMix, which results in reduced communication at the cost of computation. Our approach follows two steps. First, we compute the k powers of each shared secret, $[\![m^2]\!], \ldots, [\![m^k]\!]$ from just $[\![m]\!]$. Surprisingly, we show how to achieve this using only $O(1)$ communication per shared secret, our protocol for computing powers may be of independent interest. The second step, inspired by Ruffing et al. [70], is to to use Newton's Identities [63] to solve a system of equations of the form $S_i = m_1^i + \ldots + m_k^i$.

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

Computing powers with constant communication. For each secret share $[\![m]\!]$ sent by clients, we need to compute $[\![m^2]\!], [\![m^3]\!], \ldots, [\![m^k]\!]$. The naïve way is to directly use Beaver triples $k-1$ times. If we cared only round complexity, we could also use the constant-round unbounded fan-in multiplication [35], though it adds a 3x factor of additional work. In either case, we'd need to reconstruct $O(k)$ elements in total.

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

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

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

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

Based on Equation (1), in Figure 7, we give pseudocode for an efficient algorithm to output all the powers $[\![m^2]\!], \ldots, [\![m^k]\!]$ by memoizing the terms $[\![m^i r^j]\!]$. The algorithm requires a total of $k^2/2$ multiplications and k^2 additions in the field. The memory requirement for the table can be reduced to $O(k)$ by noticing that when we compute $[\![m^i r^j]\!]$, we only need monomials of degree $i+j-1$, so we can forget the terms of lower degree. Table 2 summarizes the asymptotic communication and computation costs of each approach.

Solving Newton's Identity. We now discuss how to reconstruct the shuffled values from the power sums. We have $S_j = \sum_{i=1}^{k} m_i^j$ where m_i is the message provided by client C_i. So we require an algorithm to extract the message m_i from S_i.

MPC Program compute-powers

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

```
Initialize Array[$k+1$][$k+1$]
for $i$ from 1 to $k$: Array[0][$i$] := $[\![b^i]\!]$
$C$ := Open($[\![m]\!] - [\![b]\!]$)
for $\ell$ from 1 to $k$: // compute all Array[$i$][$j$] where $\ell = i + j$
  sum := 0
  for $i$ from 1 to $(\ell - 1)$, $j = \ell - i$:
    sum += Array[$i-1$][$j$]
    // Invariant: sum = $\sum_{k<i} [\![m^{i-1-k} b^{j+k}]\!]$
    Array[$i$][$j$] = $[\![b^{i+j}]\!] + C \cdot$ sum
    // Invariant: Array[$i$][$j$] will store $[\![m^i b^j]\!]$ by (1)
for $i$ from 2 to $k$ output $[\![m^i]\!]$ := Array[$i$][0]
```

Figure 7: Algorithm for calculating k powers of input $[\![m]\!]$ using preprocessing in the Powermix online phase

Assuming that our goal is to mix k messages $m_1, m_2, m_3, \ldots, m_k$, the servers first run Algorithm 7 to compute the appropriate powers. Then all servers calculate $[\![S_j]\!] = \sum_{i=1}^{k} [\![m_i^j]\!]$ locally and then publicly reconstruct each S_j.

Let $f(x) = a_k x^k + a_{k-1} x^{k-1} + \ldots + a_1 x + a_0$ be a polynomial such that $f(x) = 0$ has roots $m_1, m_2, m_3, \ldots, m_k$. And we have $a_k = 1$ given that it is the coefficient of x^k resulting from the product of $(x-m_1)(x-m_2)\ldots(x-m_k)$. According to Newton's identities [70], we can calculate all coefficients of $f(x)$ by:

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

Knowing S_i we can recover all a_i by solving these equations one by one. Once we know the coefficients of $f(x)$ we can then find k roots of $f(x) = 0$ with $O(k^2)$ computation complexity in our implementation [19]. Then we recover all m_i. Our final mixing set consists of these k messages.

To conclude, Figure 8 shows the overall protocol of Powermixing.

5.3 AsynchroMix Offline Phase Requirements

The offline phase supporting AsynchroMix needs to be able to generate the requisite preprocessing elements for both converting client inputs into secret sharings and for realizing either mixing program. Of these, handling client inputs is the most straightforward as it only requires generating a t-shared random value for each input. For simplicity, we note that the randomness extraction protocol is just RanDouSha, but with only one matrix operation performed and with half the number of inputs and outputs. We, therefore, write randomness extraction as simply half of a call to RanDouSha.

```
MPC Program power-mix
• Input: $[\![m_1]\!], [\![m_2]\!], \ldots, [\![m_k]\!]$,
• Output: a shuffling of $(m_1, m_2, \ldots, m_k)$
• Precompute: $k$ sets of precomputed powers, for $k$ instances of
  compute-powers
  (i.e., $[\![b_i^j]\!]$ for $i \in [1..k], j \in [1..k]$,
  $k$ beaver triples
• Procedure:
- (Step 1) for $i$ from 1 to $k$:
    Run compute-powers (Algorithm 7) on $[\![m_i]\!]$ to obtain
    $[\![m_i^2]\!], [\![m_i^3]\!], \ldots, [\![m_i^k]\!]$
- (Step 2) for $j$ from 1 to $k$:
    Locally compute $[\![S_j]\!] := \sum_{i=1}^{k} [\![m_i^j]\!]$
    $S_i :=$ Open($[\![S_j]\!]$)
- (Step 3) Apply Newton's identities to solve $(S_1, S_2, \ldots, S_k)$, re-
  covering a shuffling of $(m_1, m_2, \ldots, m_k)$.
```

Figure 8: Power-mixing protocol for shuffling and open secret shared values $[\![m_1]\!], \ldots, [\![m_k]\!]$

Table 3: Offline phase requirements to run AsynchroMix t+1 times

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

Running our mixing programs requires additional preprocessing inputs. The Switching-Network program requires the generation of random selector bits as well as the Beaver triples needed to use them. Meanwhile, our PowerMix program needs k secret-shared powers of the same random value. These preprocessing costs are given in terms of invocations of RanDouSha and BatchRecPub in Table 3.

5.4 Supporting Larger Messages

We have so far assumed that each client message consists of a single 32-byte field element, but AsynchroMix can easily be adapted to support larger (fixed-size) messages of multiple field elements each. Since the switching network choices depend only on the preprocessed selection bits, we can simply apply the same selection bits to each portion of input (i.e., the 1st element of clients' messages are permuted in the same way as the 2nd element, and so on). For PowerMix, we could reserve a portion of each message element (e.g., $\kappa = 40$ bits) to use as a tag which would be used to link parts of a message together. Since no information about mixing inputs is leaked until the mix is opened, tags will not collide except for with $2^{-\kappa}$ probability.

6 IMPLEMENTATION AND EVALUATION

We have developed a prototype implementation that includes all of the protocols needed to realize both the offline and online phases of AsynchroMix. Our prototype is written primarily in Python 3, although with computation modules written in C++ (to use NTL [72]).[5] For batch computations on secret sharings, both the FFT-based and matrix-based algorithms are implemented in C++ using the NTL library. We carried out a distributed benchmarking experiment with several aims: to validate our analysis, to demonstrate the practicality of our approach, and to identify bottlenecks to guide future improvement. We are mainly concerned with two performance characteristics: cost and latency. Latency is the user-facing cost, the time from when the user initiates a message to when the message is published. Computation and bandwidth costs are a complementary metric since we can improve latency by adding more resources, up to the extent that sequential computations and communication round trips are unavoidable. We are mainly interested in configurations with varying the mix size k, as well as the number of servers n (and assuming $n \approx 3t + 1$). We evaluated not only the *online phase* of the MPC protocols, but also the *offline phase* which generates precomputed Beaver triples, powers, and bits.

6.1 Microbenchmarks for Robust Reconstruction

Evaluating FFT-based and matrix- based decoding. For the switching network, the main cost in the online phase is batch reconstruction. We implemented two variations of the batch reconstruction operation, one based on matrix multiplication (superlinear) as in HyperMPC [8] and others, and an alternative based on FFT (quasilinear time).[6] The use of FFT-based methods has been suggested by Damgärd et al. [41], but to our knowledge it has not been included in any MPC implementation. We give a detailed explanation of the FFT-based algorithms we use in the Appendix. Clearly for some large enough value of n, FFT-based methods would lead to a performance improvement, but we want to determine if it could provide benefits for the network sizes in our experiments.

Figure 9 shows the results of microbenchmarks for a single-core C++ implementation of the reconstruction algorithms, using a single `t2.medium` node for a series of 144 batch reconstructions of 4096 shares each, corresponding to a run of the switching network program for mixing $k = 4096$ client messages. The primary crossover point is at around $n = 2048$. ***For network sizes of*** $n = 2048$ ***and larger, FFT-based methods offer a significant (greater than 2x) improvement.*** For context, while our distributed experiment only goes to $n = 100$, HyperMPC [8] ran with up to $n = 1000$, hence the $n = 2048$ could be considered within a practical range.

We noticed that NTL switches strategies for matrix multiplication at $n = 70$. Hence at $n = 64$ the FFT evaluation performed

[5]https://github.com/initc3/HoneyBadgerMPC
[6]A function $f(n)$ is quasilinear if $f = O(n \log^c n)$ for some constant c.

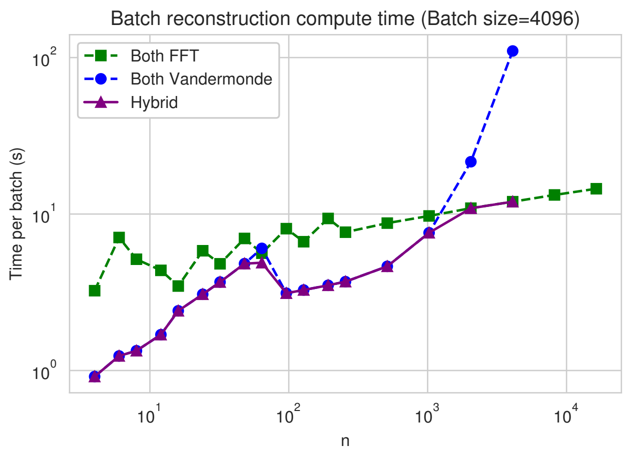

Figure 9: Compute costs for switching network application at $k = 4096$ (144x batch reconstructions of 4096 shares each) using FFT vs. Matrix Multiplication algorithms

marginally better (a 23.5% speed up) using the hybrid approach compared to just using Vandermonde matrix-based interpolation and evaluation at $n = 64$. Similarly, at $n = 1000$, the performance is close, but using FFT for evaluation but Vandermonde matrices for interpolation offers an overall benefit compared to either.

Establishing the feasibility of error correction. We implemented two algorithms for Reed Solomon error correcting, Berlekamp-Welch and Gao [47]. For up to $n = 100$, correcting errors for a single polynomial requires less than 1 second. The overall performance of the MPC system is not too dependent on the cost of error correction, because we only apply the error correction once per faulty party. Once an error is identified in any batch, we discard all the other shares from that party, and resume batch interpolation using the remaining parties. ***Hence even in the worst case where $t = 33$ servers fail sequentially, the maximum delay added would be under 33 seconds.***

6.2 Distributed Experiment for AsynchroMix

To evaluate the performance of AsynchroMix and identify the trade-offs and bottlenecks involved in our two mixing approaches, we deployed our prototype on clusters of AWS `t2.medium` instances (2 cores and 4GB RAM) in 10 regions across 5 continents. We conducted baseline tests for bandwidth and latency between instances in different regions, which we detail in Appendix B. For each experiment, we ran three trials for each configuration of n and k, and recorded the bandwidth, and running times.

Online Phase for PowerMix. Figure 10 (solid lines) shows the running time for PowerMix to mix and open from $k = 64$ to $k = 1024$ messages on up to $n = 100$ server nodes. It takes around 5 seconds to mix $k = 256$ messages on $n = 100$ servers and around 130 seconds to mix $k = 1024$ messages. We can see that PowerMix is mostly insensitive to the size of n, since the bottleneck is the computational costs, which depend mostly on k. Besides the computation steps could be parallelized to make use of more computation resources.

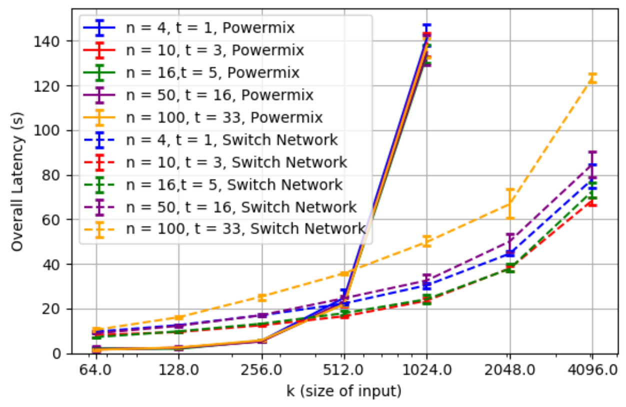

Figure 10: Online phase latency for varying number of client inputs, using PowerMix or Switching Network.

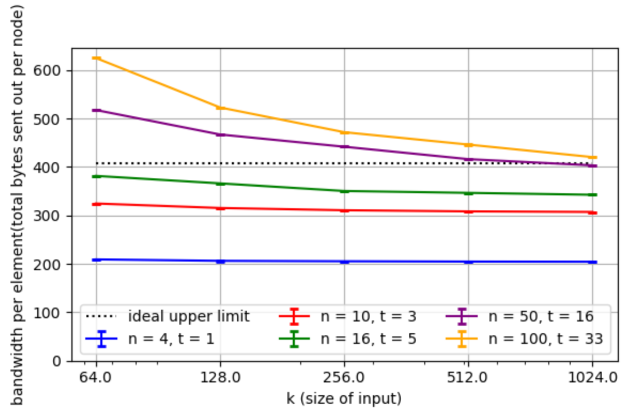

Figure 11: Communication cost (per node) of PowerMix in distributed experiment. Dashed line indicates the predicted limit as $\frac{2n}{t+1}$ approaches 6.

Figure 11 shows the communication cost of PowerMix, measured as outgoing bytes sent by each server, amortized per each client input. Since PowerMix requires two batch reconstructions of k shares each, and BatchRecPub has a linear asymptotic communication overhead to open a linear number of shares, we expect the per-server per-share cost to reach a constant for large enough n and k. We estimate this constant (the dashed line in the figure) as $2 \cdot 6 \cdot 1.06 \approx 12\times$, where the 2 is for the two batch reconstruction instances used in PowerMix, 6 is the is the overhead for each batch reconstruction (the limit approached by $\frac{2n}{t+1}$), and 1.06 is the observed overhead of Python `pickle` serialization in our implementation. As n grows larger, since there is an additive overhead quadratic in n, larger values of k are necessary for the amortization to have effect. However, even at $n = 100$, only around 400 bytes are needed to mix each 32-byte message with $k = 512$ or higher.

Online Phase for Switching Network. Figure 10 (dashed lines) shows the running time for Switching Network to mix from $k = 64$ to 4096 messages. We can shuffle $k = 4096$ messages on $n = 100$

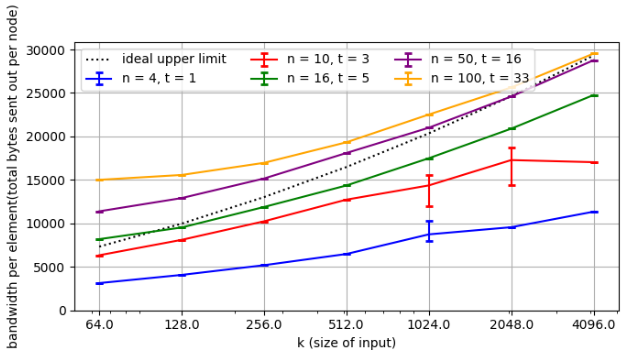

Figure 12: Communication cost (per node) of switching network in distributed experiment. Dashed line indicates the predicted limit as $\frac{2N}{t+1}$ approaches 6.

servers in around 2 minutes. Since the number of batch reconstruction rounds grows with $\log^2 k$, the sensitivity to n also increases as k increases.

Based on the microbenchmarks (Figure 9), at $k = 4096$ and $n = 100$, the inherent computation time should account for only about 3 seconds out of the total 120 seconds observed. The rest is due to a combination of serialization and Python overhead as well as communication. Fig 12 shows the overall communication cost of the Switching network. For $k = 4096$ client inputs with $n = 100$ servers, each input requires each server to transmit nearly 30 kilobytes. The dashed line here is $y = 32 \cdot 6 \cdot \log^2 k$ where 6 is reconstruction overhead and $\log^2 k$ corresponds to the number of total rounds. From our baseline experiment, the worst per-instance bandwidth is 221Mbps (São Paolo) and the longest round trip latency is 328ms (São Paolo to Mumbai), hence up to 50 seconds can be explained by transmission time and latency. Hence at this setting, computation, and communication contribute about equally (neither is the sole bottleneck), although there appears to a considerable room to eliminate overhead due to serialization and Python function calls in our implementation.

Tradeoffs between PowerMix and Switching Network. In the online phase, PowerMix requires considerably more computation but less communication than Switching Network. Given the resources available to our `t2.medium` instances, PowerMix results in more than 2× reduction in overall latency at $n = 100$ for up to $k = 512$ clients, but for larger values of k, Switching Network is preferable. PowerMix would naturally be useful for larger values of k in more bandwidth-constrained or computationally-powerful networks.

Overall cost for AsynchroMix. Figures 13 and 14 show the estimated overall cost, per server and per client input, combining both computation ($0.05 per core hour for an EC2 node) and bandwidth ($0.02 per gigabyte transferred out) costs based on AWS prices. The stacked bar charts show the costs broken down by phases (offline, online, and client input). The offline phase contributions are based on a distributed experiment for the RanDouSha algorithm, multiplied out by the necessary number of preprocessing ingredients of each type (see Table 3). The offline cost of PowerMix is always

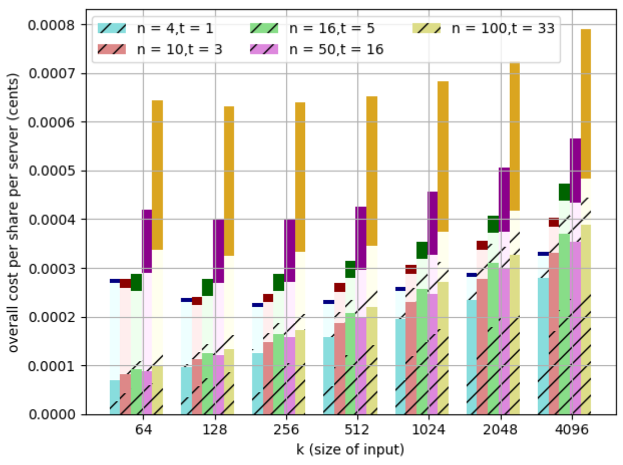

Figure 13: Estimated combined cost (computation and bandwidth) for AsyncMix with Switching Network. The cost includes offline phase cost(dark colored), online cost(light colored) and client input cost(top). Bandwidth cost is marked as "//".

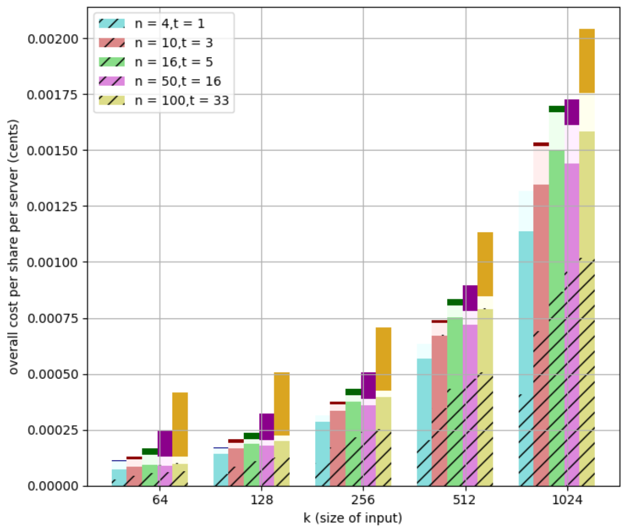

Figure 14: Estimated combined cost (computation and bandwidth) for AsyncMix with PowerMix. The cost includes offline phase cost(dark colored), online cost(light colored) and client input cost(top). Bandwidth cost is marked as "//".

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

7 CONCLUDING REMARKS

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

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

In the future, our effort should motivate other MPC implementations to consider robustness as well as a computation vs communication trade-off.

Acknowledgements

We thank Ashish Choudhury and Arpita Patra for their valuable comments, as well as Sylvain Bellemare for software development. This work has been partially supported by the National Science Foundation under grant CNS-1719196; by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network; and by the Department of Energy's Office of Cybersecurity, Energy Security, and Emergency Response and the Department of Homeland Security's Security Science & Technology Directorate under Award Number DE-OE0000780.

REFERENCES

[1] Masayuki Abe and Fumitaka Hoshino. 2001. Remarks on mix-network based on permutation networks. In *International Workshop on Public Key Cryptography (PKC)*. 317–324.

[2] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2018. Validated Asynchronous Byzantine Agreement with Optimal Resilience and Asymptotically Optimal Time and Word Communication. arXiv preprint arXiv:1811.01332.

[3] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. 2017. MCMix: Anonymous Messaging via Secure Multiparty Computation. In *USENIX Security Symposium*. 1217–1234.

[4] A Aly, M Keller, E Orsini, D Rotaru, P Scholl, NP Smart, and T Wood. 2019. *SCALE–MAMBA v1. 3: Documentation.* Technical Report.

[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *ACM EuroSys.*

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

[7] Michael Backes, Amit Datta, and Aniket Kate. 2013. Asynchronous computational VSS with reduced communication complexity. In *CT-RSA*. 259–276.

[8] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. 2018. An End-to-End System for Large Scale P2P MPC-as-a-Service and Low-Bandwidth MPC for Weak Participants. In *ACM CCS*. 695–712.

[9] Donald Beaver. 1991. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology – Crypto*. 420–432.

[10] Zuzana Beerliová-Trubíniová and Martin Hirt. 2008. Perfectly-secure MPC with linear communication complexity. In *Theory of Cryptography Conference*. 213–230.

[11] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *ACM STOC*. 1–10.

[12] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous secure computations with optimal resilience. In *ACM PODC*. 183–192.

[13] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from Bitcoin. In *IEEE Symposium on Security and Privacy.*

[14] The Tor Blog. [n.d.]. One cell is enough to break Tor's anonymity. https://blog.torproject.org/blog/one-cell-enough. Accessed Nov 2018.

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

[16] Dan Bogdanov, Riivo Talviste, and Jan Willemson. 2012. Deploying secure multiparty computation for financial data analysis. In *International Conference on Financial Cryptography and Data Security*. 57–64.

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

[18] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. 2007. Denial of service or denial of security?. In *ACM CCS*. 92–102.

[19] Alin Bostan, Laureano Gonzalez-Vega, Herve Perdry, and Eric Schost. [n.d.]. Complexity issues on Newton sums of polynomials.

[20] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.

[21] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains.* Ph.D. Dissertation.

[22] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. 2002. Asynchronous Verifiable Secret Sharing and Proactive Cryptosystems. In *ACM CCS*. 88–97.

[23] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology—Crypto*. 524–541.

[24] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *IEEE Symposium on Reliable Distributed Systems*. 191–201.

[25] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[26] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. 2018. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *Advances in Cryptology – Crypto*. 34–64.

[27] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. 2017. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *ACM CCS*. 719–728.

[28] Ashish Choudhury, Martin Hirt, and Arpita Patra. 2013. Asynchronous multiparty computation with linear communication complexity. In *DISC*. 388–402.

[29] Ashish Choudhury, Emmanuela Orsini, Arpita Patra, and Nigel P Smart. 2016. Linear Overhead Optimally-Resilient Robust MPC Using Preprocessing. In *SCN*. 147–168.

[30] Ashish Choudhury and Arpita Patra. 2015. Optimally resilient asynchronous MPC with linear communication complexity. In *ICDCN*. 5.

[31] Ashish Choudhury and Arpita Patra. 2017. An efficient framework for unconditionally secure multiparty computation. *IEEE Transactions on Information Theory* (2017), 428–468.

[32] Sandro Coretti, Juan Garay, Martin Hirt, and Vassilis Zikas. 2016. Constant-round asynchronous multi-party computation based on one-way functions. In *International Conference on the Theory and Application of Cryptology and Information Security*. 998–1021.

[33] Henry Corrigan-Gibbs and Bryan Ford. 2010. Dissent: accountable anonymous group messaging. In *ACM CCS*. 340–350.

[34] Artur Czumaj and Berthold Vöcking. 2014. Thorp shuffling, butterflies, and non-Markovian couplings. In *International Colloquium on Automata, Languages, and Programming*. 344–355.

[35] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*. 285–304.

[36] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. 2008. Asynchronous Multiparty Computation: Theory and Implementation. Cryptology ePrint Archive. https://eprint.iacr.org/2008/415.

[37] Ivan Damgård and Yuval Ishai. 2005. Constant-round multiparty computation using a black-box pseudorandom generator. In *Advances in Cryptology—CRYPTO*. 378–394.

[38] Ivan Damgård, Yuval Ishai, and Mikkel Krøigaard. 2010. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Advances*

in Cryptology—EUROCRYPT. 445–465.

[39] Ivan Damgård, Yuval Ishai, Mikkel Krøigaard, Jesper Buus Nielsen, and Adam Smith. 2008. Scalable multiparty computation with nearly optimal work and resilience. In *Annual International Cryptology Conference.* 241–261.

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

[41] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and unconditionally secure multiparty computation. In *Advances in Cryptology—CRYPTO.* 572–590.

[42] D. Das, S. Meiser, E. Mohammadi, and A. Kate. 2018. Anonymity Trilemma: Strong Anonymity, Low Bandwidth Overhead, Low Latency - Choose Two. In *IEEE Symposium on Security and Privacy (SP).* 108–126.

[43] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-generation Onion Router. In *USENIX Security Symposium.*

[44] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *ACM CCS.* 2028–2041.

[45] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[46] Fabienne Eigner, Matteo Maffei, Ivan Pryvalov, Francesca Pampaloni, and Aniket Kate. 2014. Differentially private data aggregation with optimal utility. In *ACSAC.* 316–325.

[47] Shuhong Gao. 2003. A new algorithm for decoding Reed-Solomon codes. In *Communications, Information and Network Security.* 55–68.

[48] Daniel Genkin, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. Privacy in decentralized cryptocurrencies. *Commun. ACM* 61, 6 (2018), 78–88.

[49] Yossi Gilad. 2019. Metadata-Private Communication for the 99%. To appear in Commun. ACM.

[50] S Dov Gordon, Feng-Hao Liu, and Elaine Shi. 2015. Constant-round MPC with fairness and guarantee of output delivery. In *Advances in Cryptology—CRYPTO.* 63–82.

[51] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *IEEE Symposium on Security and Privacy (SP).*

[52] Ryan Henry, Amir Herzberg, and Aniket Kate. 2018. Blockchain Access Privacy: Challenges and Directions. *IEEE Security & Privacy Magazine* 16, 4 (2018), 38–45.

[53] Martin Hirt and Jesper Buus Nielsen. 2006. Robust multiparty computation with linear communication complexity. In *Advances in Cryptology—CRYPTO.* 463–482.

[54] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8.

[55] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS.* 830–842.

[56] Marcel Keller, Valerio Pastro, and Dragos Rotaru. 2018. Overdrive: making SPDZ great again. In *Adanvance in Cryptology—EUROCRYPT.* 158–189.

[57] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. 2016. Fair and robust multi-party computation using a global transaction ledger. In *Advances in Cryptology—ASIACRYPT.* 705–734.

[58] Ranjit Kumaresan and Iddo Bentov. 2014. How to use bitcoin to incentivize correct computations. In *ACM CCS.* 30–41.

[59] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. 2017. Atom: Horizontally Scaling Strong Anonymity. In *SOSP.* 406–422.

[60] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. 2016. Riffle. *Privacy Enhancing Technologies* (2016), 115–134.

[61] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.

[62] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis. In *OSDI.* 711–725.

[63] John D. Lipson. 1976. Newton's Method: A Great Algebraic Algorithm. In *3rd ACM Symposium on Symbolic and Algebraic Computation.* 260–270.

[64] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. 2004. Fairplay-Secure Two-Party Computation System.. In *USENIX Security Symposium.*

[65] Fabio Massacci, Chan Nam Ngo, Jing Nie, Daniele Venturi, and Julian Williams. 2018. FuturesMEX: secure, distributed futures market exchange. In *IEEE Symposium on Security and Privacy (SP).* 335–353.

[66] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. 2013. Zerocoin: Anonymous distributed e-cash from bitcoin. In *IEEE Symposium on Security and Privacy.* 397–411.

[67] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *ACM CCS.* 31–42.

[68] Udaya Parampalli, Kim Ramchen, and Vanessa Teague. 2012. Efficiently shuffling in public. In *International Workshop on Public Key Cryptography.* 431–448.

[69] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The loopix anonymity system. In *USENIX Security Symposium.* 16–18.

[70] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. 2017. P2P Mixing and Unlinkable Bitcoin Transactions. In *NDSS.*

[71] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[72] Victor Shoup et al. 2005. NTL, a library for doing number theory.

[73] Nigel P Smart and Tim Wood. 2019. Error Detection in Monotone Span Programs with Application to Communication-Efficient Multi-party Computation. In *CT-RSA.* 210–229.

[74] Alexandre Soro and Jérôme Lacan. 2010. FNT-based Reed-Solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference.* 1–5.

[75] Shi-Feng Sun, Man Ho Au, Joseph K Liu, and Tsz Hon Yuen. 2017. RingCT 2.0: A compact accumulator-based (linkable ring signature) protocol for blockchain cryptocurrency Monero. In *European Symposium on Research in Computer Security.* 456–474.

[76] Yixin Sun, Anne Edmundson, Laurent Vanbever, Oscar Li, Jennifer Rexford, Mung Chiang, and Prateek Mittal. 2015. RAPTOR: Routing Attacks on Privacy in Tor. In *USENIX Security Symposium.* 271–286.

[77] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A Distributed Metadata-Private Messaging System. In *26th SOSP.* 423–440.

[78] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Global-scale secure multiparty computation. In *ACM CCS.* 39–56.

[79] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. 2013. Hang with your buddies to resist intersection attacks. In *ACM CCS.* 1153–1166.

[80] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: a general-purpose compiler for private distributed computation. In *ACM CCS.* 813–826.

A BATCH SECRET SHARING WITH QUASILINEAR COMPUTATION

Damgärd et al. [38, 39] first suggested the use of FFT-based operations for batch secret sharing, although to our knowledge this has never been implemented previously. We would naturally expect quasilinear operations to be necessary when scaling n to extreme large networks. However, even at the smaller values of n up to 100 that we consider, we investigated whether FFT-based operations could offer performance improvements.

A.1 Shamir Sharing in FFT-friendly fields

In Section 2 we give a description of Shamir sharing and batch operations for arbitrary prime-order field $\mathbb{F}_p$, and for arbitrary evaluation points α_i. To enable FFT-based operations, we choose $\mathbb{F}_p$ such that $2^\kappa | p-1$, and hence we can find a 2^κ-th root of unity, ω. Concretely, in our implementation we choose p as the order of the BLS12-381 elliptic curve, such that $2^{32}|p-1$, and $p \approx 255$ bits.

A.2 Batch secret share operations using FFT

Given a polynomial $\phi(\cdot)$ in coefficient form, it is clear how to use FFT to evaluate it at points ω^i for $i < n$. The offline phase makes use of randomness extraction. As mentioned in Section 2, the standard approach is to perform multiplications by a hyperinvertible matrix multiplication, such as the Vandermonde matrix. By choosing the Vandermonde matrix defined by $\alpha_i = \omega^i$, this can be evaluated efficiently using FFT.

As defined in Section 5, Robust-Interpolate depends on a subroutine to interpolate a polynomial from an arbitrary subset of $t+1$ shares. Soro and Lacan [74] give a transformation that relies on several FFTs and is quasilinear overall. Soro and Lacan's approach has a setup cost of $\mathcal{O}(n\, log^2\, n)$ which depends on the points we are interpolating from, and a cost of $\mathcal{O}(n\, log\, n)$ per interpolation after that. More specifically, the cost per interpolation consists of a standard inverse FFT and a polynomial multiplication which is done using an FFT/CRT based approach by NTL. In A.4 we give a detailed explanation of this method.

If the first attempt at decoding $2t+1$ received shares fails, we know there is at least one error, but we don't know where it is. With each additional value we wait for, we either identify the error, or learn the number of errors is one more, in which case we wait for an additional point. This is known as Online Error Correction [28]. We implement Gao's algorithm for Reed Solomon decoding, which is $O(n \log n)$ when using using FFT for polynomial multiplication.

A.3 Vandermonde interpolation

Given $t+1$ points $((x_0, y_0), (x_1, y_1), \dots, (x_t, y_t))$ for distinct values $(x_0, x_1, \dots, x_t)$, polynomial interpolation means finding the lowest degree polynomial $P(X)$ such that $P(x_i) = y_i$. In general, given $t+1$ points we can always find such a polynomial that is of degree at most t. Lagrange interpolation is the standard algorithm used for polynomial interpolation,

$$P(X) = \sum_i^t \left(y_i \prod_{j \neq i}^t \frac{X - x_j}{x_i - x_j} \right). \tag{2}$$

However, this has a quadratic computational cost of $O(t^2)$, and is impractical for large t. An alternative approach to interpolation, as in HyperMPC [8] for example, is to use matrix multiplication with the inverse Vandermonde matrix, M^{-1}, where $M_{i,j} = x_i^j$. To summarize:

Step 1 (depends only on $x_0, \dots, x_t$):

- Compute the inverse of M^{-1}

Step 2 (depends also on $y_0, \dots, y_t$):

- Matrix multiply $(a_0, \dots, a_t)^T = M^{-1}(y_0, \dots, y_t)^T$ such that $P(X) = \sum_i a_i X^i$.

To interpolate a batch of k polynomials at once, we multiply M^{-1} by a matrix of size $\{t+1\} \times k$.

A.4 FFT-based interpolation

Here we give a self-contained explanation of the FFT-based polynomial interpolation algorithm from Soro and Lacan [74]. In this setting we assume the additional constraint that each x_i is a power of ω, a primitive n^{th} root of unity,

$$x_i = \omega^{z_i} \qquad z_i \in \{0, 1, \dots, n-1\}$$

The goal is to get an expression for $P(X)$ that can be computed within $O(n \log n)$ steps depending on $y_0, \dots, y_t$, along with a precomputation phase depending only on $x_0, \dots, x_t$. We start by rewriting Equation (2) as

$$P(X)/A(X) = \sum_i^t \frac{y_i / b_i}{X - x_i}. \tag{3}$$

where we define

$$A(X) = \prod_j^t (X - x_j), \tag{4}$$

and

$$b_i = \prod_{j \neq i}^t (x_i - x_j) = \frac{A(x_i)}{x_i - x_j}. \tag{5}$$

The degree-t polynomial $A(X)$ as well as each b_i depends only on $\{x_i\}$ and so we compute them explicitly during an initialization phase. The right hand side is intractable to compute directly, but we can make use of the Taylor series expansion $1/(X - x_i) = -\sum_j x_i^{-j-1} X^j$. We therefore have

$$P(X)/A(X) = -\sum_i^t \left(\sum_j^t (y_i/b_i) x_i^{-j-1} X^j \right) \mod X^{t+1} \tag{6}$$

Rearranging, we have

$$P(X)/A(X) = -\sum_j^t \left(\sum_i^t (y_i/b_i) x_i^{-j-1} \right) X^j \mod X^{t+1} \tag{7}$$

and finally since $x_i = \omega^{z_i}$, we can replace each coefficient with a polynomial evaluation

$$P(X)/A(X) = -\sum_j^t N(\omega^{-j-1}) X^j \mod X^{t+1} \tag{8}$$

where we define the polynomial

$$N(X) = \sum_i^t (y_i/b_i) X^{z_i}. \tag{9}$$

To summarize, we can compute $P(X)$ through the following steps:

Step 1 (depends only on $x_0, \dots, x_t$):

- Compute $A(X)$, $\{b_i\}$.

Step 2 (depends also on $y_0, \dots, y_t$):

- Compute $N(X)$ from coefficients $\{y_i/b_i\}$.
- Evaluate each $N(\omega^j)$ using FFT to obtain the coefficients of $P(X)/A(X) \mod X^{t+1}$.
- Multiply by $A(X)$ to recover $P(X)$.

For interpolation of a batch of k polynomials from shares received from the same set of $t+1$ parties, Step 1 can be computed once based on the party identifiers. Soro and Lacan [74] give an algorithm to compute this step in $O(n \log^2 n)$ overall time. Step 2 can clearly be computed in $O(n \log n)$ time, and must be computed for each of polynomial in the batch.

A.5 Microbenchmarks

We now perform microbenchmarks to evaluate when FFT-based methods are more performant than Vandermonde matrix multiplications. We consider the following tasks and algorithms:

Task	$\approx O(n^{1+c})$	$\approx O(n \log^c n)$
Encode Shares	Matrix Mul	FFT
Interpolate	Matrix Mul	Soro-Lacan [74]
RSDecode	Berlekamp-Welch	Gao

We implemented all algorithms in C++ using the NTL library. Additional details on costs for interpolation, evaluation, matrix inversions, etc and on methodology are given below.

Timing evaluation algorithms: The core component of evaluation using Vandermonde matrices is multiplication of a $n \times (t+1)$ matrix and a $(t+1) \times k$ matrix, where k is the number of polynomials to evaluate. We use NTL for matrix multiplication. We set $k = 8192$ to be large enough to estimate the amortized cost per evaluated polynomial. For FFT-based evaluation, the operation

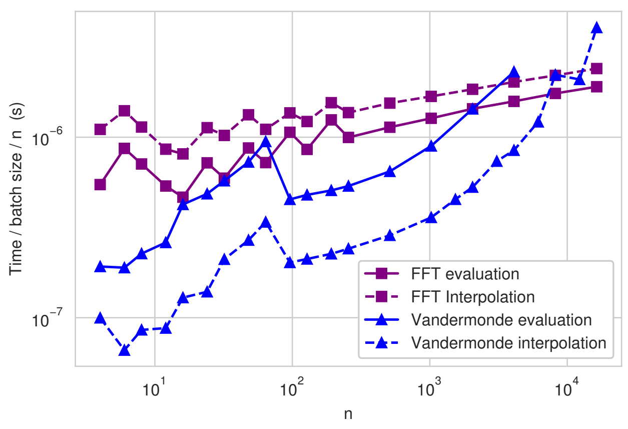

Figure 15: Interpolation (Step 2) and Evaluation Micro-Benchmarks

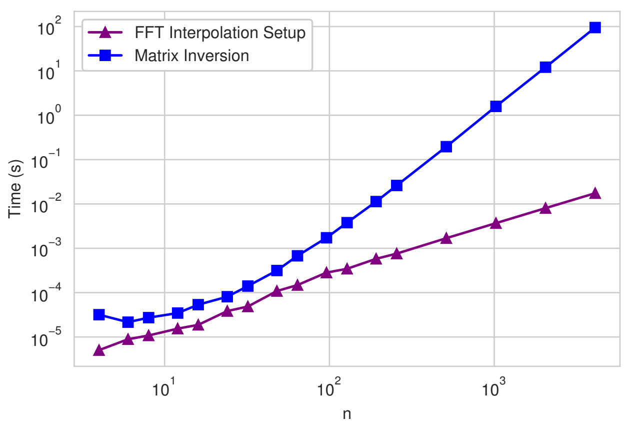

Figure 16: Interpolation preparation (Step 1) time micro-benchmarks

consists simply of an FFT applied to each of the k polynomials in turn. Figure 15 shows the costs of these components.

Timing interpolation algorithms: The interpolation algorithms both have a setup phase which only depends on the x-coordinates of the points we are interpolating on. In the context of batch reconstruction, these coordinates only depend on the first $t + 1$ parties we received shares from. Therefore, the setup phase only needs to be done once within a single round of batch reconstruction. The primary component of the interpolation algorithms are also dependent on the batch sizes. We time these two parts of all algorithms separately which helps us accurately predict how our execution time would vary with both n and the batch size.

Vandermonde-based interpolation and evaluation costs roughly $\mathcal{O}(n^2)$, while their FFT-counterparts take $\mathcal{O}(n \log n)$ time. However, FFT has a relatively large constant behind the big-O notation but is only better than Vandermonde-based operations at relatively larger values of n ($n \geq 8192$). When the costs for matrix inversion, as shown in Figure 16, are included in the total costs, in practice we see a cross-over much earlier since matrix inversion.

Total cost for batch reconstruction: Our current implementation of batch reconstruction requires 3 evaluations and 2 interpolations. Additionally, we perform batch size$/(t+1)$ evaluations / interpolations per batch. Therefore, the total cost of a single batch reconstruction is given by

$$\begin{aligned} 2 \times \text{Cost per interpolation} \times \\ \text{batch size}/(t+1) + 3\times \\ \text{Cost per evaluation} \times \text{batch size}/(t+1) \end{aligned}$$

B DETAILS ON DISTRIBUTED EXPERIMENT SETUP

To launch distributed experiments on both Powermix and Swtiching Network, we set up AWS machines in up to 10 regions across 5 continents around the world. We tested the performance of both methods in the following settings : $n = 4, n = 10, n = 16, n = 50, n = 100$ and corresponding region settings are recorded in Table 4.

Regions	n = 4	n = 10	n = 16	n = 50	n = 100
Virginia	1	1	2	5	10
Ohio	0	1	1	5	10
Oregon	0	1	2	5	10
Frankfurt	0	1	1	5	10
Tokyo	1	1	2	5	10
Mumbai	1	1	1	5	10
South America	1	1	2	5	10
Canada	0	1	1	5	10
London	0	1	2	5	10
Paris	0	1	2	5	10

Table 4: Table of Region Setting for AsynchroMix Online Phase Benchmark (n is the number of peers)

For a better understanding of the network situation among different AWS nodes, we launched tests to measure the latency and bandwidth among AWS peers in different regions. The result of latency experiment could be found at Table 5 and we measured it by letting peers ping each other. With the help of $iperf3$, we managed to measure the per link bandwidth among the peers. The result of bandwidth experiment is available in Table 6. Besides per link bandwidth, we also get total outgoing bandwidth which are measured when all peers communicate with all other peers. Total outgoing bandwidth provides a better view of actual communication and benchmark result is available in Table 7.

Regions	Virginia	South America	Tokyo	Frankfurt	Canada	Paris	Ohio	Oregon	London	Mumbai
Virginia	X	145	162	91.2	16.4	81.6	11.6	79.8	75.9	187
South America	145	X	271	233	123	221	151	184	213	328
Tokyo	162	271	X	241	154	234	155	100	236	129
Frankfurt	91.1	233	241	X	99.1	19.6	101	155	12.8	133
Canada	16.4	123	154	99.1	X	93.9	25.6	65.1	85.8	196
Paris	81.5	221	234	10.6	93.9	X	92.3	153	8.56	106
Ohio	11.6	151	155	103	25.6	92.7	X	70.2	85.9	196
Oregon	79.7	184	100	155	65.2	152	70.1	X	141	224
London	75.9	213	237	12.8	85.9	8.52	86	141	X	114
Mumbai	187	328	129	113	196	106	196	224	114	X

Table 5: Latency tests of AWS machines across different regions. (round trip time in ms, instance type: t2.medium)

Regions	Virginia	South America	Tokyo	Frankfurt	Canada	Paris	Ohio	Oregon	London	Mumbai
Virginia	X	38.6	39.6	72.7	159	35.6	200	94.2	48.9	23.7
South America	46.4	X	28	28.2	63.8	25	60.2	27.6	25.4	17.4
Tokyo	33.4	22.9	X	32.6	33	22.6	45.1	35.4	25.7	36.8
Frankfurt	42.6	25.3	32.6	X	56.1	114	56.6	28.4	196	43.1
Canada	116	60.4	52.1	54.2	X	62	280	45.3	67.5	32.9
Paris	36.1	23.9	18.9	433	56	X	115	61.9	335	34.9
Ohio	104	45.6	38	61	92.5	42.8	X	52.9	54	28.7
Oregon	56.9	35.3	60.8	46.8	87.2	39.3	91.7	X	47.4	29.4
London	58	30.7	25.4	300	51.1	600	70.9	66.1	X	43.9
Mumbai	22.6	15	50.2	71.5	29.9	43	31.3	23	45.7	X

Table 6: Per link bandwidth test of AWS machines across different regions (per link bandwidth in Mbps, instance type: t2.medium)

Regions	Total Outgoing Bandwidth (Mbps)
Virginia	618.5
South America	221.5
Tokyo	236.2
Frankfurt	487.2
Canada	529
Paris	377.65
Ohio	450.5
Oregon	259.38
London	305.4
Mumbai	401.1

Table 7: Overall bandwidth test for AWS machines across different regions (total outgoing bandwidth in Mbps, instance type: t2.medium)