ZIPNet: Low-bandwidth anonymous broadcast from (dis)Trusted Execution Environments

Michael Rosenberg
University of Maryland
College Park, MD, USA
micro@umd.edu

Maurice Shih
University of Maryland
College Park, MD, USA
maurices@umd.edu

Zhenyu Zhao*
Tsinghua University
Beijing, China
jasonzhao404@gmail.com

Rui Wang*
Purdue University
West Lafayette, IN, USA
rui.wang.rw683@yale.edu

Ian Miers
University of Maryland
College Park, MD, USA
imiers@umd.edu

Fan Zhang
Yale University
New Haven, CT, USA
f.zhang@yale.edu

ABSTRACT

Anonymous Broadcast Channels (ABCs) allow a group of clients to announce messages without revealing the exact author. Modern ABCs operate in a client-server model, where anonymity depends on some threshold (e.g, 1 of 2) of servers being honest. ABCs are an important application in their own right, e.g., for activism and whistleblowing. Recent work on ABCs (Riposte, Blinder) has focused on minimizing the bandwidth cost to clients and servers when supporting large broadcast channels for such applications. But, particularly for low bandwidth settings, they impose large costs on servers, make cover traffic costly, and make volunteer operators unlikely.

In this paper, we describe the design, implementation, and evaluation of ZIPNet, an anonymous broadcast channel that 1) scales to hundreds of anytrust servers by minimizing the computational costs of each server, 2) substantially reduces the servers' bandwidth costs by outsourcing the aggregation of client messages to untrusted (for privacy) infrastructure, and 3) supports cover traffic that is both cheap for clients to produce and for servers to handle.

1 INTRODUCTION

Anonymous communication is increasingly important for dissent, activism, and even finance. The simplest form of anonymous communication, anonymous broadcast, is commonly envisioned as a Twitter-like platform for free speech. But anonymous broadcast also forms a critical component of larger cryptographic systems such as e-cash [15], differentially private telemetry [17], and even single secret leader election [10]. These systems require comparatively low bandwidth, especially compared to increasingly high-bandwidth residential Internet.[1] Without the former constraints on client-side bandwidth, we observe that there is a missing dimension in the design of anonymous broadcast systems that can now be tackled: *trust diversity*.

Anonymity loves company, but modern anonymous broadcast systems make that company prohibitively expensive on two fronts:

To boost anonymity set size, we need *cover traffic* — empty messages that are indistinguishable from "real" traffic. Luckily, cover traffic is in theory cheap, since cover users do not contend for space to send messages, we need not increase the size of the broadcast channel. However, state-of-the-art designs often impose high costs for cover traffic, require trusted parties to handle bandwidth costs for cover traffic and pay concretely high computational costs for extra traffic even if it is from cover users. They also impose concretely high computational costs on users who submit cover traffic messages.

Second, "anonymity loves company" extends not just to the company of other honest users, but to the company of honest servers: modern anonymous broadcast systems rely on some number N_S of servers to run the broadcast system, where some number $t \leq N_S$ must be honest to provide anonymity. But if we are limited (by theory or practice) to, e.g., $N_S \leq 2$, the scheme is not much better than a simple trusted third party: compromising the two operators breaks anonymity for every user of the system.

To ensure trust diversity, we want N_S to be as large as possible to decrease the marginal trust put in each server operator. This is not just a question of developing cryptography that supports $N_S > 2$ servers. In practice, we need servers to be cheap enough to instantiate that the *operators* of the servers can be selected from a diverse set of entities, e.g., volunteers (as in Tor) or non-profit organizations. Many anonymous broadcast schemes, however, incur very high computational costs that make such trust diversity unlikely if not impossible.

In light of these observations, we revisit the design choices for anonymous broadcast systems through the lens of the *anonymity trilemma*: strong anonymity, low latency, and low bandwidth, pick at most two [21].

Strong anonymity is essential. The lack of strong network layer anonymity can defeat the privacy provided at other layers of a system. Take the case of Zerocash [8], a protocol for privacy-preserving payments whose deployed derivatives include Zcash [27], Tornado Cash [36], and other cryptocurrencies. Zero-knowledge proofs hide the source of spent funds (and hence the payer's identity) in the set of all past transactions. But payment privacy also requires we hide network metadata, e.g., the payer's IP address, when broadcasting an anonymous transaction to the blockchain. Here we hit a snag: common network anonymity schemes, e.g., Tor [22], can only hide client IP addresses amongst a small number of concurrent network users and are subject to traffic analysis attacks [6]. The net result is that network anonymity becomes the limiting factor in the privacy of a Zerocash-like system. Similarly, the statistical guarantees of differentially private schemes in the shuffle model [9][2] are only as

*Part of work done while at Yale

[1] The median US fixed-connection upload bandwidth in 2019 was 10Mbps [25].

[2] The shuffle can be, as in Prio [17], instantiated with an anonymous broadcast channel

Table 1: Notational legend for common DC net parameters

Notation	Description
M	# users (talking and non-talking)
N	# talking users ($\ll M$)
N_S	# servers
$\|m\|$	bit length of a single message slot
B	bit length of the entire broadcast message

strong as the quality of the shuffle, i.e., the anonymity of the submitted results and the trust diversity of the servers. Strong network anonymity reduces these cross-layer privacy mismatches.

Security vs latency should not be a tradeoff. In many anonymous messaging systems, adding additional servers comes at a latency penalty. Consider mixnet and mixnet-like schemes such as Mixminion [31], Loopix [35], or Vuvuzela [38], where each additional server increases the latency of the broadcast functionality. This raises two problems. First, for latency-sensitive applications, there's some maximum amount of trustworthy anonymity achievable — systems may support perhaps 3 parties. But more subtly, even latency-tolerant applications face tradeoffs: does the cost of additional latency justify the addition of an additional party? While latency is a concretely measurable cost, the additional privacy provided by a volunteer server is hard to quantify. Worse, anonymous messaging systems need to maximize user participation over groups with disparate utility (e.g., latency) vs privacy tradeoffs. Ideally, additional trusted parties should come at no cost.[3]

(Client-side) bandwidth is not a concern. The size of the broadcast channel for many applications (such as anonymizing cryptocurrency transactions) is small, especially relative to increasing residential broadband capacity. For Ethereum concretely, this is about 6-7 kilobytes a second.[4]

The missing dimension: server bandwidth and resource usage. As mentioned above, a key component to trust diversity is lowering the cost of participating in an anonymous broadcast system both as a user providing cover traffic and, more crucially, as a server. Existing modern anonymous broadcast systems substantially increase the costs of adding additional anytrust servers and increase the cost of cover traffic to those servers.

1.1 DC nets revisited

In this work, we focus on *dining-cryptographer networks*, or *DC nets*, and related schemes that we broadly refer to as *anonymous vector schemes* . These systems classically offer the strongest anonymity and lowest latency of the anonymous broadcast systems. The modern construction of these systems, first introduced in Dissent [19], operates in a client-server model, with a number of *anytrust* servers performing cryptographic operations on a vector of messages submitted by clients, to produce an anonymous broadcast message. As long as at least one server is honest, anonymity is guaranteed (hence, *any* trust).

The core of anonymous vector schemes is, effectively, a one-time pad. Suppose a system with M users permits $N \ll M$ users to send $|m|$-bit messages concurrently (a broadcast channel of size $|m|N$) in a single *round*. Each client generates a vector of the same size, comprised of the XORs of multiple [5] of random pads, each of which is derived from a shared key with an anytrust server. If transmitting, clients XOR their message into a slot in the vector. If not transmitting, the client message is effectively 0s. Each server XORs of all received messages and further XORs in all of its shared one-time pads. Finally, the servers XOR their partial results to derive the broadcast message. As far as communication costs, each client submits a message of $|m|N$ and each server receives $|m|NM$ bits each round.

Compression schemes. Recent work on anonymous vector schemes , such as Blinder [2], Riposte [18], and Express [24], which we refer to broadly as *compression schemes*, use cryptography to reduce client messages from $O(|m|N)$ down to $O(|m|\sqrt{N})$ or $O(|m| \log N)$ in the two server case. When client bandwidth is limited relative to the size of the broadcast channel, compression schemes are invaluable.

Compression schemes, however, come at a cost: by drastically increasing the cost of operating an anytrust server and of handling cover traffic. Schemes such as Riposte and Express do not practically (or at all for Express) support more than 2 servers. Even for schemes that do support more servers, such as Blinder, the cost of running a server is likely prohibitive to volunteer server operators, requiring $O(|m|N)$ asymmetric cryptographic operations per client message. Worse, the cost of each additional server[6] imposes considerable costs on every client in terms of additional computational overhead when preparing message shares for each server.

Our Contribution. In this paper, we propose ZIPNet, a DC net-like anonymous vector scheme that alleviates these trust diversity problems and offers a 4.2x-7.6x reduction in server runtime compared to the state of the art [3]. ZIPNet accomplishes this by relying on trust for everything but anonymity. In particular, ZIPNet outsources handling almost all of the server inbound traffic and roughly 80 percent of computation cost, to third-party servers not trusted for privacy. While this approach is fully compatible with traditional DC nets, ZIPNet greatly simplifies the design and client computational costs by relying on client-side trusted execution environments for DoS prevention but, crucially, not for privacy.

ZIPNet separates a DC net into three distinct components: 1) clients that author messages, 2) aggregators, who compress messages as they traverse a network of untrusted (for anonymity) servers, and 3) servers, who provide anonymity via the same any-trust mechanism as Dissent. By separating the DC-net into distinct components with differing security requirements, we can narrowly define each component's security needs and optimize.

For clients, the key to our approach is the observation that client DC nets can rely on trusted hardware for DoS prevention *without* harming anonymity. As observed in [37], TEEs, when used as a

[3] This is, of course, a tunable trade-off between trust and availability. Secret sharing can be used to trade off trust in a given party for availability if they go offline.

[4] Block size is a limit on the capacity of the network to accept transactions, but strictly speaking, more transactions can be submitted. Some amount of buffer for anonymous broadcast would be necessary.

[5] Some schemes use addition in fields of larger characteristics.

[6] In the two-server case, DPF schemes are cheap, using a tree-based PRF constructed from e.g., hashes. These techniques do not generalize.

form of zero-knowledge proof, no longer need to maintain the confidentiality of data (from the prover/TEE operator). Rather, they must merely ensure correctness. We go one step further, using TEEs as a falsifiable trust assumption: an attacker who breaks the TEE can DoS the system by submitting malformed messages, but this will be observable. Moreover, because an attacker who breaks our TEE assumption only extracts their own secrets, TEE failures do not harm privacy. TEEs give us an extremely lightweight means to ensure message integrity and correct behavior. We avoid expensive traitor tracing protocols or verifiable shuffles.

Second, by separating the roles of handling messages from the role of servers providing anonymity, we can substantially reduce the bandwidth costs for operating anytrust servers compared to Blinder, Express, or even Dissent. Aggregators can compress messages multiple client messages together by XORing them. These work as a type of reverse content distribution network (CDN). Instead of processing *all* client messages, each of size B, anytrust servers take only a *single* aggregate message of size B, plus the list of user IDs in the anonymity set.

Better yet, because these aggregator servers are untrusted for privacy, they can be run as infrastructure by a single party without undermining security or trust diversity.[7]

Third, ZIPNet specifically minimizes the cost of cover traffic to anytrust servers, a point typically neglected in other systems, but essential for anonymity. The size of the broadcast vector is determined by the number of active users, but the number of messages each server must process is the sum of the active users and the cover users. In Dissent and standard DC nets, anytrust server bandwidth is thus $O(BM)$. Compression schemes can reduce this by a factor of $O(|m| \log N)$ or $O(|m|\sqrt{N})$, but they substantially increase the computational cost of processing any message: for every message, each anytrust server must do asymmetric cryptographic work that is proportional to B, the size of the broadcast vector.[8]

In ZIPNet, we get both the low computational costs of DC-net style anytrust servers, requiring only symmetric operations per client message plus a signature check on a constant size hash, and reduced anytrust server bandwidth that is even smaller than with compression schemes. By outsourcing the aggregation step to completely untrusted (for privacy) infrastructure, volunteer anytrust servers pay almost no additional bandwidth cost for each additional cover traffic message and very low computational costs.

Finally, for the concrete instantiation of ZIPNet, we target the setting where client bandwidth substantially exceeds the size of the broadcast bandwidth but clients do not necessarily know when they will transmit (e.g. to make a payment). As such, the use of TEEs allows us to implement an incredibly simple reservation mechanism that gives low latency for unscheduled messages (in contrast, in schemes like Dissent, clients must schedule in advance, and the scheduling process typically has high latency). We develop a scheme that is reminiscent of the basic collision avoidance mechanisms in, e.g., Ethernet, and inspired by footprint scheduling [29].

[7] In contrast, one cannot simply pay for servers that are trusted for anonymity to get trust diversity. Many naïve approaches leave nodes controlled by a single coordinated party, and decentralized approaches can rapidly lead to all infrastructure being hosted on the cheapest or simplest cloud computing providers, leading to an increasingly central point of compromise.

[8] Either directly to evaluate the DPF or, in the case of Blinder, in the setup for the MPC that checks the secret-shared DPF.

Table 2: Communication complexity of different anonymous broadcast protocols, where λ is the security parameter, and we assume the broadcast channel size $B = N|m|$, up to a constant factor.

	User	Anytrust server	Cost to add non-talker
DC Net [16]	$O(NM\|m\|)$	-	-
Anytrust [19]	$O(N\|m\|)$	$O(N\|m\|(M+N_S)))$	$O(N\|m\|)$
Blinder [2]	$O(\sqrt{N}\|m\|)$	$O(\|m\|(M\sqrt{N}+NN_S))$	$O(\sqrt{N}\|m\|)$
Riposte [18]	$O(\|m\|)$	$O(NM\|m\|)$	-
ZIPNet (this paper)	$O(N\|m\|)$	$O(M\lambda + NN_S\|m\|)$	$O(\lambda)$

In summary, we introduce ZIPNet which offers:

- Lightweight DoS prevention from falsifiable trust assumptions with TEEs.
- anonymous message aggregation: using untrusted (for privacy) aggregators, we can aggregate all client messages into a single message to reduce bandwidth overhead of anytrust servers.
- high performance: Concretely, ZIPNet is 4.2x-7.6x faster than the state-of-the-art schemes. The bandwidth overhead incurred by each server for each additional cover traffic message is only 84 bytes.
- Low-cost trust diversity: each additional anytrust server is cheaper to operate than the state-of-the-art schemes and imposes far smaller costs on clients sending messages.

2 ARCHITECTURE

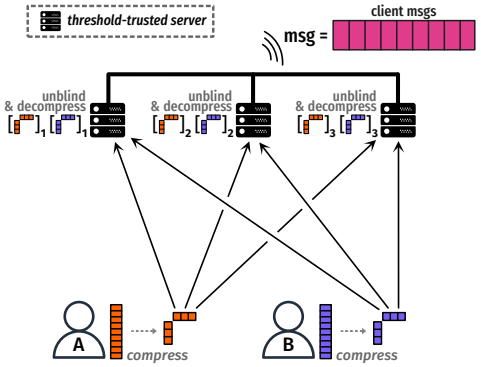
In this section we present the architecture of ZIPNet.

2.1 Overview of DC net architecture

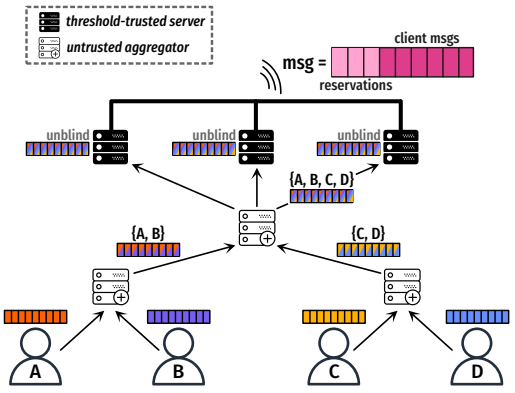
Before diving into how ZIPNet works, we review common DC net constructions. A DC net is typically run by a set of M users and optionally some servers. The ideal anonymity a DC net can offer is M-anonymity, meaning from the adversary's point of view, the probability that a given broadcast message is sent by any particular user is $1/M$. In the original definition [16], only one user talks (we use *speak*, *talk*, and *send* interchangeably), and the remaining $M-1$ users help increase the anonymity set by sending cover traffic (all-zero messages). A DC net protocol can also allow $N \ll M$ users to talk simultaneously using a scheduling mechanism to share the bandwidth. We denote the bit length of a single user message slot as $|m|$, and the bit length of the entire broadcast channel as B. We provide the notation as a stand-alone figure in Table 1.

Chaum's DC net [16]. The dining cryptographers' problem and its first solution (a DC net) are presented in Chaum's seminal paper [16]. To realize a broadcast channel of bandwidth B, the total communication complexity is $O(BM^2)$ as it involves all-to-all communication among M users. Note that the communication complexity is independent of N, the number of users that are actually speaking.

Specifically, users first establish pair-wise keys so that user i and j will share a key $k_{i,j}$. To send, an active user i first computes a B-bit one-time pad (OTP) $k_i = \oplus_{j \neq i} k_{i,j}$. Then she composes



(a) An anonymous broadcast round in a scheme similar to Blinder [2]. A user compresses their message into the outer product of two $\sqrt{N}$-sized vectors, which are then split into secret shares and sent to the majority-trust servers.



(b) An anonymous broadcast round in ZIPNet. Ciphertexts are XORed through multiple aggregation layers, until the root aggregator broadcasts the aggregate and user list to all servers. The servers then unblind and combine the aggregates into the final broadcast message.

Figure 1: Diagrams representing the structures of two anonymous broadcast systems

a B-bit message msg by embedding her message (of size $|m|$) at proper locations according to the schedule and putting zeroes in all other positions. User i then broadcasts msg $\oplus\, k_i$. Users who are not scheduled to send will still broadcast a zero message (encrypted with their respective OTPs) to provide cover traffic. In total, for a single message, each server sends B bits to $M-1$ other server, thus, the global communication cost is $O(BM^2)$ bits.

DC net with servers. Dissent [19] uses a group of servers to avoid expensive all-to-all broadcasts among users. In the setup, users establish shared keys with N_S servers, of which, the server needs only trust one for privacy. To broadcast in a system with just one talking user, each user sends an $|m|$-bit message (properly constructed according to the schedule) to an anytrust server encrypted with an OTP derived from shared keys with all servers. The servers XOR all received messages with their keys to decrypt and deliver the round message. The same generalization applies, wherein the broadcast channel can be of any size B that can accommodate the scheduling and talking of N users.

In this architecture, all-to-all communication among users is avoided—users just send B bits to the server. On the other hand, anytrust servers still need to consume BM bytes, plus the communication overhead among them. Other works, such as Riposte [18] and Blinder [2] perform message compression on the client side, enabling quadratic or even exponential improvements in per-message communication cost.

The DC net with servers paradigm also has a choice of trust assumptions. While Dissent has an anytrust model, Blinder operates in a *majority-trust* model, i.e., at least $N_S/2$ servers must be honest in order to maintain privacy.

We include a graphical representation of schemes like Blinder in Figure 1a, using quadratic compression and a majority trust assumption.

2.2 ZIPNet overview

ZIPNet extends the anytrust model with several key modifications. First, we introduce *aggregator nodes* to reduce the bandwidth overhead of anytrust servers. Second, we use trusted execution environments (TEEs) to prevent denial of service (DoS) attacks that are otherwise expensive to prevent.

There are three types of participants in ZIPNet: *clients*, *aggregators*, and *anytrust servers*. We call clients who send (nonzero) messages *talking clients*, and those who send cover traffic (zero messages) *non-talking clients*.

Every client is assigned a single aggregator, and clients operate from inside a TEE. The clients, aggregators, and anytrust servers all check the attestation signatures of network parties they are peered with to ensure that all relevant parties are acting from within TEEs. This ensures that all parties in the network are running the same "acceptable" code base, which prevents malicious clients and aggregators from causing a denial of service, by, e.g., writing in other clients' slots. As we will see, ZIPNet's privacy guarantees do not rely on the security of the TEE, but rather on the honesty of at least 1 server.

The protocol proceeds in *rounds*. In each round, clients, aggregators and anytrust servers interact to produce a *broadcast message* as follows:

Clients A talking client produces a ciphertext of their message (formatted according a scheduled write slot) and sends it to their designated aggregator. A non-talking client provides cover traffic by choosing a message of all-zeros.

Aggregators Upon receiving ciphertexts from clients, aggregators temporarily store them, aggregate them by XORing them together, and then forward the aggregate message to the next upstream aggregator. The top-level *root* aggregator will produce a single B-bit vector and send it to the anytrust servers. Aggregators can significantly reduce the

traffic that anytrust servers must ingest: messages from M users can be aggregated to a single message of size $O(B)$. Upon first glance, this does not seem to really change the nature of the problem, as the aggregators may become the new bottleneck. However, in ZIPNet, we make aggregators *untrusted*, so there can be many of them to distribute the workload.

Anytrust servers Upon receiving an aggregate ciphertext, anytrust servers partially decrypt the ciphertext with the keys they have. To conclude the round, they combine the partial decryptions to obtain the broadcast message and publish it.

3 DESIGN AND SECURITY ANALYSIS

The core procedures are given in Algorithms 1 to 3. These algorithms are executed by clients, aggregators, and any trust servers, respectively, in the configuration shown in Fig. 1b. We operate in the any trust model introduced by [19] and used in subsequent works[2, 18], where servers (in our case aggregators and any trust servers) are picked statically in advance at setup and we assume at least one of the any trust servers is honest for anonymity and all of the servers are honest for availability. Algorithm 1, which runs inside a TEE, is executed by clients to produce messages. The trusted execution environment ensures that clients follow the protocol and prevents DoS attacks. Messages are passed to aggregators and then to any-trust servers as shown in Fig. 1b. These later two algorithms are run outside of TEEs with security coming from the any-trust model. Below we detail design decisions and some details for system setup and rate limiting elided from the figures exposition.

Falsifiable TEE assumption. A TEE is a hardware-based isolated execution environment that aims to provide confidentiality and integrity guarantees to enclosed code and data. While TEEs can help simplify and speed up protocols, practical TEE implementations may fail to provide any security guarantees [11–13]. As a result, applications of TEEs are faced with a binary choice—they either fully trust TEEs and swallow the considerable risk of side-channel attacks, or reject TEEs altogether and forgo their benefits.

With ZIPNet, we propose a new way to use TEEs where trust assumptions are falsifiable and breaks are recoverable. A key design consideration is that *ZIPNet only relies on TEEs for liveness* and that *TEE failures are conspicuous in ZIPNet.* An attacker who breaks a TEE does not compromise confidentiality, rather they merely gain the ability to DoS the system by spamming malformed messages. This is detectable by all parties. Further, since each client 's TEE contains no sensitive information, recovery from a TEE break consists of patching the TEE and re-registering clients.

Realizing a falsification protocol for disruptive malicious behavior is surprisingly simple: clients (running in TEEs) are required to append a falsification tag to their messages. This tag consists of hash of their message. Any malicious client who, after breaking their own TEE, submits a malformed message will either a) write to an unused slot or b) collide with an honest client's message in a used slot. In the latter case, some bits of the honest client's message and/or hash will be flipped. If we assume the hash behaves as a random oracle, then with overwhelming probability, these flips will not result in a valid (message, tag) pair. As tag generation is required by the enclave, a "false" alarm where a malicious client intentionally submits a bad tag for their own message would also require an enclave break.

Setup. We assume a PKI where all parties have registered. In the case of clients, registration includes an attestation proving that their secret key (a Curve25519 scalar in our implementation) is controlled by the enclave. Naïvely, every party in ZIPNet could perform TEE attestation for every protocol message. However, Intel SGX's attestation mechanism includes extra overhead: every verification performs a network request to Intel to check for revocation. To avoid this repeated cost, we require parties to generate a signing key upon setup precisely once and provide an attestation for its correct generation and sealedness. Thus, following setup, parties can simply sign their communications with that key rather than performing a full attestation.

A client's secret key (kept private in the enclave) is used for key agreement and signing. A user obtains the server's public keys from the PKI and provides them to the enclave to derive shared secrets using the standard Diffie Hellman key exchange. As shown in Algorithm 1, the enclave will derive one sharedKeyWithServer for each server and store them sealed in ssDB. For each round, a fresh OPT is derived as line 19-20 in Algorithm 1 using a KDF (HKDF in our implementation). Shared keys are symmetrically ratcheted [4] (i.e., new keys are obtained by applying KDF to old keys) for forward secrecy, as line 23 of Algorithm 1. Severs also perform the same ratcheting operation so the shared keys remain consistent.

Sealed data. To store a value persistently, the TEE *seals* it, i.e., encrypts it with a key that's locked inside the TEE, and makes the ciphertext available to the operating system. To recover the value, the TEE caller provides the sealed ciphertext and a reference to the key that decrypted it. Care is required to ensure that a stateful protocol using a TEE model is not susceptible to tampering around the boundaries: while the TEE state cannot be modified due to cryptographic guarantees on the sealing process, *which* state (of all previous states) is presented at a given protocol step is up to the (possibly malicious) caller. Careful consideration must be given to rollback attacks. In particular, in ZIPNet, all behavior within a TEE is deterministic, with randomness derived from fixed keys and unique inputs tied to the round.

3.1 Weak TEEs, rate limiting and fair resource usage without trusted state

ZIPNet is designed to support very large amounts of cover traffic in a setting where a small number of users dynamically send messages. As such, we cannot rely on some fixed transmission schedules as in, e.g., Dissent. At the same time, precisely because the broadcast channel is relatively small, we need to ensure malicious clients cannot monopolize it. In other words, we want quick transmission and free cover traffic.

If the client's TEE is trusted to maintain state, enforcing a rate limit is trivial: a counter that resets at the end of each round is incremented every time the client sends a real (non-cover) message and the client attests $\mathsf{ctr} < \mathsf{limit}$. But many TEEs, including Intel SGX for servers, cannot prevent rewinding of ctr or another state. Moreover, fault injection attacks have plagued many secure hardware implementations that do offer state-keeping.

To avoid the state-keeping problem, ZIPNet uses a simplification of rate-limiting tags from [14]. For each message, the client outputs $\mathsf{PRF}_k(\mathsf{ctr}\|\mathsf{epoch})$ and attests that $\mathsf{ctr} < \mathsf{limit}$. Any attempt to rewind the counter will result in a duplicate tag detected by the Aggregators. For cover messages, the client generates a completely random tag independent of the counter, thus avoiding the rate limit. We elide the details from our pseudocode for readability, but the implementation is straightforward and included in our prototype of ZIPNet.

3.2 Scheduling

As with any DC net system, ZIPNet users can only talk in a reserved message slot. ZIPNet uses footprint scheduling [29] and piggybacks on the broadcast channel to include a scheduling message at the very beginning (i.e., round r of the protocol also schedules the slots for round $r + 1$). To prevent a malicious user from depleting the channel by spamming reservations, ZIPNet runs the scheduling in the TEE. Again, even if a TEE is breached, the adversary can only disrupt the scheduling (liveness failure), not privacy.

Suppose a user U wishes to speak in round $r + 1$, she reserves a slot in round r as follows. First, she computes a slot using a pseudorandom function $s = \mathsf{PRF}(k, 0, r) \mod \mathsf{numSchedSlots}$ and an f-bit *footprint* $F = \mathsf{PRF}^f(k, 1, r)$ where k is U's secret key. She speaks in round r, and writes F in slot s. Now, at the conclusion of round r, she detects collision by comparing the s-th slot in the broadcast message with F. If they are different, then someone else tried to schedule the same slot, and she simply tries again the next round. Setting f to be sufficiently long can ensure that collisions are detected with overwhelming probability. If F does match the broadcast message at slot s, then she uses the (signed) broadcast message of round r as a ticket to her TEE that will allow her to speak in round $r + 1$.

Offline clients. Clients need to transmit a reservation request and empty message first. Once they have a reservation, they can transmit a message (and additional requests as needed). Clients who go offline restart this process.

Integrity. ZIPNet runs two anonymous broadcast channels in parallel. One for messages and one to reserve slots in the next round. However, an attacker who can equivocate the schedules and feed each client a distinct one can deanonymize users. While the schedule is signed in ZIPNet, this is part of the anti-client-DoS machinery and is done by a single server that is not trusted for privacy.

One possible solution to address this concern would involve having all servers sign the scheduling vector for each round. This approach would result in additional network overhead due to the necessity of coordinating signatures. Instead, in ZIPNet, both clients and servers rely on the schedule as a shared context during the derivation per round key material. If a client and server disagree on the output of the preceding round, the result will be a random message vector.

Scheduling vector size. Naively, we could set the scheduling vector to have a one-to-one mapping to the scheduling of the message vector. However since slots are selected at random, this would needlessly reject clients. Based on the calculations made by Riposte [18], when m users write randomly to a single slot, in order to have 95% non-collision rate, there need to be $2.7m$ slots. We set our scheduling vector to contain $4m$ slots while maintaining a message vector of m slots. This gives us an approximate 97% non-collision rate for the message vector.

Support users with high network latency. The current scheduling algorithm works under the assumption that a user's network latency to ZIPNet is lower than the round time. For users with high network latency variance, we can extend the algorithm to allow reservations valid for more than one round. This would accommodate users who need multiple rounds to schedule. The rate limit tokens can be adapted accordingly (e.g., by including $i||i{+}1||\cdots\|i{+}t$ in the PRF computation where t is the number of rounds in which a user is scheduled to speak).

3.3 Security analysis

ZIPNet relies on TEEs for integrity protection from clients and trusts both the anytrust servers and aggregators for availability and integrity. We discuss deployment settings and how to improve availability in Section 6. Here we consider the anonymity of ZIPNet.

Adversary and network model. We adopt the standard adversary and network model of anytrust DC nets (e.g., as in Blinder [2]), assuming there are at least ρN honest clients, for some $\rho \in (0, 1)$, submitting their messages, granting the network adversary the power to spawn $(1 - \rho)N$ clients. The adversary can inspect, but cannot block, all network channels, as no protocol can guarantee meaningful privacy if the global network adversary can drop all honest messages.

ZIPNet is a round-based protocol and assumes synchrony. Anonymity will not be breached when this assumption is violated (liveness will). Synchrony assumption has proven practical in real-world distributed systems such as blockchains.

We consider a modified version of the definition proposed in Riposte [18] for anonymity. In addition to syntactic changes to support our scheme and simplifying the definition to consider a single honest server, we make one important modification: we remove the direct requirement in the game that there are two honest clients. Instead, we limit the number of accounts the attacker can control. As a result, the definition requires the honest server to enforce a minimum participation threshold in a round and abort if the attacker drops. If it fails to do so, the game is trivially winnable. This is a key requirement for practical security in many schemes and one that many definitions fail to enforce.

A (t, n)-anonymous broadcast system is defined by a security game between a challenger who operates $t - n$ honest clients and one honest server, and an adversary who operates the remaining parties including an aggregator, the remaining servers, and at most n dishonest clients. For each honest client, the adversary specifies both their message m and the slot s they should write to. The challenger picks a bit b and based on it either computes client messages as specified or permutes which client writes which message. Given the computed client messages, the adversary provides a claimed aggregation and the challenger responds with the honest server's round output given the aggregate. The full definition is given in Appendix A

Security argument. We now sketch the argument for the anonymity of a stand-alone DC net with aggregation, and then our DC net with scheduling. We make no security assumptions about TEEs.

We note that, without aggregation or scheduling, the protocol is a standard DC net in the exact anytrust model of Dissent. Security stems from the fact that each client message is indistinguishable from random given a single honest server. As a result, the adversary cannot learn which client wrote which message or the bit b. While client messages are malleable, there is a one-to-one correspondence between bits in any client message and the corresponding bit in the broadcast vector, so flipping a bit merely flips the same bit in the broadcast.

At first glance, aggregation gives the adversary another attack vector not present in classic DC nets: it can drop or alter messages. However, this is not new: an attacker who controls the network can already both drop and modify messages in standard DC nets. As in Dissent, honest servers in ZIPNet must abort a round if they receive less than a critical number of client messages.

ZIPNet, however, is not a single DC net. Its two DC nets, specifically (t, n)-anonymous broadcast systems, run in parallel. In round i the client sends a message in the scheduling network. In round $i + 1$ the client sends in the transmission network if they see their reservation in the broadcast schedule. First, we note that revealing the schedule or even letting the attacker control it does not impact privacy: our definition of anonymous broadcast anonymity assumes the attacker can control which slots are written to. Second, not sending the schedule to a particular client is equivalent to dropping the client's messages, which is already guarded against by the minimum participation threshold. However, per the above discussion on scheduling integrity, an attacker could equivocate on the schedule. Using the schedule as a shared KDF input for both clients and servers prevents this attack.

4 IMPLEMENTATION AND EVALUATION

In this section, we evaluate ZIPNet in multiple scenarios and applications. ZIPNet targets a regime with low message size, a small number of broadcasters per round, and a high degree of required anonymity (i.e., having a large number of non-talking clients to provide *cover traffic*). Similar to Riposte [18] and Blinder [2], anonymous microblogging is a practical application for ZIPNet. These are also the network properties needed by privacy-preserving cryptocurrencies: anonymity is important, block sizes (proportional to the number of participants per round) are small, and message (transaction) sizes are in the hundreds of bytes; high anonymity calls for increased trust diversity both to better match the strong privacy offered by cryptography on-chain and to meet users' expectations for decentralization.

ZIPNet is designed to reduce the computational overhead of clients and servers as we increase trust diversity both by adding anytrust servers and cover traffic. We find that, in our motivating regime, ZIPNet significantly outperforms Blinder on server runtime regardless of the number of servers. This is likely due to cheaper cryptographic protocols, and the concretely small bandwidth and computation necessary for an anytrust server to operate.

4.1 Implementation details

Client. We instantiate ZIPNet client with Intel SGX as the underlying TEEs in ~2.2k lines of Rust using the Teaclave SGX SDK [26]. We ported third-party crates x25519-dalek and ed25519-dalek to the SGX environment for Diffie-Hellman key exchange and digital signature respectively. For key derivation and pseudorandom number generation, we ported `hkdf` and `aes-ctr` from the RustCrypto [1].

Aggregator. We implement the ZIPNet aggregator in ~2k lines of Rust, with 16-thread multithreading.

Server. We implement the any-trust server in ~2.2k lines of Rust code. It uses the same cryptography libraries as the client, but runs outside TEEs. The majority of server computation time is spent on unblinding the message share. To accelerate the PRNG used to expand server keys to round-specific OTPs, we use hardware acceleration (AES-NI) with multithreading.

4.2 Experiment setup and design

Setup. We target a deployment model where servers will be run by community volunteers. Therefore in our experiment, we deploy any trust servers on affordable AWS EC2 instances (in particular, AWS `t2.2xlarge` with 8 vCPUs and 32GB of RAM, costing $0.3712 per hour on demand). The aggregator will be run by well-funded organization, thus we use powerful servers (in particular, AWS `c6a.8xlarge` with 32 vCPUs and 64GB of RAM, costing $1.224 per hour on demand) to run aggregators. We run the client on an OVH server (`Infra-1-LE`) with Intel SGX hardware support.

We evaluate the performance of ZIPNet in both LAN and WAN settings. In the LAN setting, all servers and aggregators are in the same AWS availability zone. In the WAN setting, servers are distributed across the globe (in US East (Ohio, N. Virginia), US West (N. California, Oregon), Canada (Central), Europe (Paris, Frankfurt, London), Asia Pacific (Tokyo), South America (São Paulo)), with the aggregator in Ohio. We use `iperf` to measure the bandwidth and latency between the aggregator and servers and the results are in Tables 4 and 5. The bandwidth ranges from 69.6Mbps to 985Mbps and latency ranges from below 1ms to 130ms. To simulate large numbers of client requests, we first pre-generate client requests on a dedicated server with Intel SGX enabled, then replay them at the aggregator. The cost of generating client requests is measured separately. In our experiments, we vary the message size, and the number of clients and servers to illustrate the characteristics of our system. Unless specified, all experiments used a single aggregator.

Design. DCnet protocols, including ZIPNet, run in fixed rounds where the round time is set statically so that the system can handle some parameterized number of talking clients (which determines the size of the broadcast channel B), non-talking (cover traffic) clients and a fixed number of any trust servers.

The goal of our experiments is to determine the minimal round time our implementation can sustain across these parameters. To do this, we modify ZIPNet aggregators and servers to instead of operating on fixed round times, immediately complete their step in a round when they receive a fixed number N (recall notations from Table 1) of messages. We then run the system end-to-end with mostly simulated client traffic (except for evaluating client performance). This gives us the latency each component of ZIPNet

Algorithm 1 Client-side procedures,includes sending a message in the current round, sending cover traffic in the current round, and reserving a slot for the next round. These procedures run inside a TEE.

Public Input
  (round, msg, requestSlot, publishedSchedule, σ_{PubSched})
Sealed State
  usk User signing key
  lpk Leader signing public key
  k User symmetric key
  ssDB Shared secrets with server (key exchange done at user registration)
Let nextSchedVec, msgVec, := $[0, 0, \ldots, 0]$
assert $\text{Verify}_{\text{lpk}}$(publishedSchedule, σ_{PubSched})
if requestSlot **then** ▷ try to reserve a slot for talking in the next round if requested
  Let (nextSchedSlot, nextFootprint) := compSchedFootprint(k, round + 1)
  Set nextSchedVec[nextSchedSlot] $\oplus$= nextFootprint

▷ Recompute the request from the last round and see if it made undisturbed into the published scheduled. If it is, we completed a reservation and can write the message to the reserved slot.
Let (curFPSlot, curFootprint) := compSchedFootprint(k, round)
if publishedSchedule[curFPSlot] = curFootprint **then**
  Let falsificationTag = ROMHash(msg)
  Set msgVec[compMsgSlot(curFPSlot, publishedSchedule)] $\oplus$= msg||falsificationTag
else ▷ If reservation failed, nothing is XOR'd, i.e., this is cover traffic
  Set msgVec[compMsgSlot(curFPSlot, publishedSchedule)] $\oplus$= 0

▷ In all cases, we blind the broadcast vectors either to send or provide cover traffic
for sharedKeyWithServer $\in$ ssDB **do**
  Let pad1||pad2 := KDF(sharedKeyWithServer, round, publishedSchedule)
  Set nextSchedVec $\oplus$= pad1 and msgVec $\oplus$= pad2
Let payload := (round, nextSchedVec, msgVec)
Let σ := Sign_{usk}(payload)
Set ssDB = ssDB.rachet()
Output (payload, σ)

procedure compSchedFootprint(k, round)
  Let slot := PRF_k(round) mod numSchedSlots
  Let footprint := PRF^f_k(round)
  return (slot, footprint)

procedure compMsgSlot(curFPSlot, publishedSchedule)
  Let msgSlot := 0
  for $i \in$ 0..curFPSlot **do**
    if publishedSchedule[i] $\neq$ 0 **then**
      Set msgSlot+ = 1
  return msgSlot

Procedures in this figure run inside a TEE

incurs to handle that many messages for a given number of talking clients, non-talking (cover) clients, and a set number of anytrust servers. From this, we can compute the total round time of ZIPNet as a whole.

4.3 Experimental results

Below we present three sets of results. As an overview, Figures 2 and 3 give latency numbers for anytrust servers and aggregators as a function of either the amount of non-talking (cover traffic) clients or the number of talking clients. In the former case, the size of the broadcast channel remains constant. In the latter, the size of the broadcast channel increases to handle the increased demand for broadcast space. Separately, Fig. 4 reports client runtime also as a function of increasing load. Since client runtime is not affected by other clients, we measure instead the cost of increased broadcast channel size (indirectly caused by more talking clients) and, separately, the cost of increasing the number of anytrust servers. I.e., the cost of increased trust diversity.

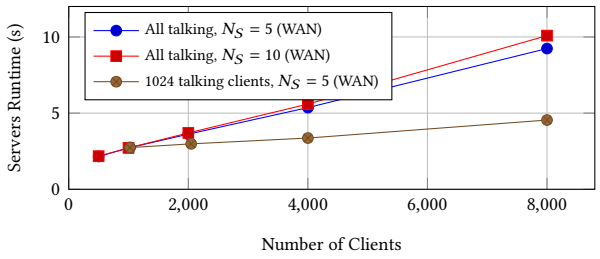
4.3.1 Anytrust server runtime. In our experiment, we fix a leader server to combine the outputs of follower servers to get the final broadcast message. Figure 2 plots the runtime of server processing, starting from when all servers receive the final aggregate, to the time when the leader outputs.

Algorithm 2 Aggregator receiving user message

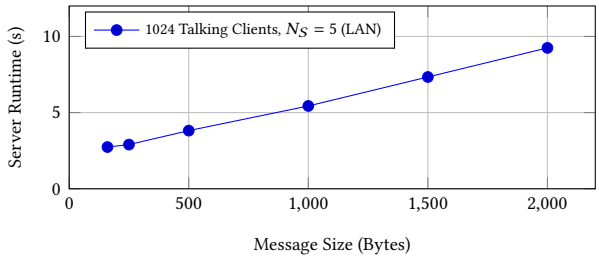
Public Input
  upk User Public Key
  σ Payload signature
  payload Payload from user
State
  ask Aggregator signing key
  aUserPKs Aggregated user public keys
  aMsgVec Aggregate of msg vector
  aSchedVec Aggregate of next schedule vector
  regUsers Set of registered users
  curRound current round number
Let (round, nextSchedVec, msgVec) := payload
assert upk $\in$ regUsers
assert $\text{Verify}_{\text{upk}}$(payload, σ)
assert upk $\notin$ aUserPKs
assert round = curRound
▷ XOR the payload into the aggregate vectors
Set aSchedVec $\oplus$= nextSchedVec
Set aMsgVec $\oplus$= msgVec
Set aUserPKs = aUserPKs $\cup$ {upk}
Let payload
  := (round, aUserPKs, aSchedVec, aMsgVec)
Let σ' := Sign_{ask}(payload)
Output (payload, σ')

Algorithm 3 Server receiving data from aggregator

Public Input
  publishedSchedule
  payload Payload from aggregator
  σ Payload signature
State
  apk Aggregator public key
  ssk Server signing key
  regUsers Set of registered users
  ssDB Shared secrets with clients
  minClients Min. allowed clients per round
assert $\text{Verify}_{\text{apk}}$(payload, σ)
Let (round, userPKs, aSchedVec, aMsgVec)
  = payload
assert userPKs $\subseteq$ regUsers
assert |userPKs| $\geq$ minClients
▷ Unblind broadcast vectors with per user keys
for userPK $\in$ userPKs **do**
  Let pad1||pad2
    := KDF(ssDB[userPK], round, publishedSchedule)
  Set aSchedVec $\oplus$= pad1
  Set aMsgVec $\oplus$= pad2
Let payload := (round, aSchedVec, aMsgVec)
Let σ := Sign_{ssk}(payload)
Set ssDB := ssDB.rachet()
Output (payload, σ)



(a) Server Runtime as Number of Clients Increase. "All talking" means all of the clients are talking, while "1024 talking" means the other clients send cover traffic. Message size is 160B.



(b) Server Runtime As Message Size Increases.

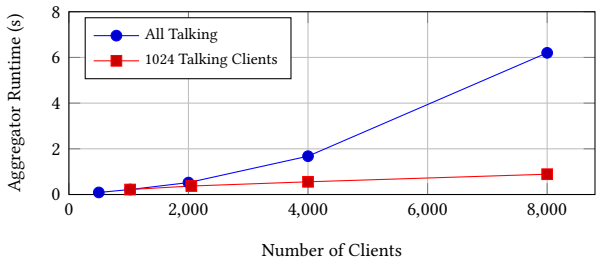
Figure 2: Server runtime

In the left figure, we give latency as the number of clients increases in two scenarios. The line labeled "1024 Talking Client" plots an increase in cover traffic for a constant number of talking clients and therefore a constant broadcast channel. The other lines plot an increase in the number of talking clients (and therefore an increase in the broadcast channel size), for 5 servers and 10 servers respectively. Two conclusions can be drawn. First, the performance difference between 5 and 10 servers is insignificant, especially with many clients, as the overhead is dominated by decrypting of messages and the traffic between servers is low (a single broadcast message). Second, the numbers confirm that server runtime increases quadratically with the number of talking clients, but only linearly with cover traffic. This highlights one of ZIPNet's advantages that adding a non-talking client is cheap.

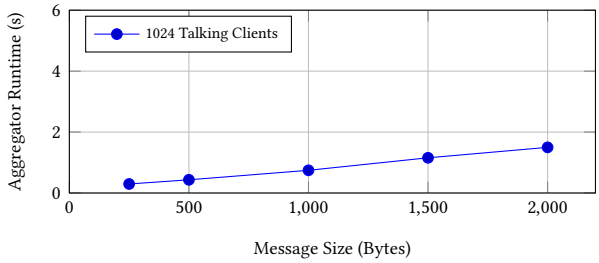
In the right figure, we measure latency as the size of messages in the broadcast channel increases but the number of clients is kept constant. This gives us a baseline for systems effectiveness when used for e.g, anonymous Twitter (message size is 280 bytes) versus an anonymous cryptocurrency (400 bytes for Bitcoin, 108 bytes for Ethereum, 2KB for Zcash, 2.38KB for Monero).

4.3.2 Aggregator runtime. An aggregator's work is to XOR M messages of size $B = bN$, so its runtime is a function of the number of talking and non-talking clients. Figure 3 plots the concrete runtimes under varying parameters.

Figure 3a plots the same experiments as Fig. 2a but measures aggregator performance. For clarity, we do not plot measurements for both 5 and 10 anytrust servers as aggregator performance is



(a) Aggregator Computational Runtime as Number of Talking Clients Increase (and Broadcast Channel Increases) or as Cover Traffic Increases. Messages size is 160B.



(b) Aggregator Computational Runtime As Message Size Increases.

Figure 3: Aggregator runtime

independent of the number of servers. Again, we can observe that the additional runtime overhead of adding a non-talking client is much cheaper than adding a talking one. Figure 3b confirms that the runtime of the aggregator is linear in the message size when the number of clients is fixed.

4.3.3 Client runtime. The performance profile of clients is different from aggregators and servers. To send, the client computes N_S pairwise OTPs of length B and XOR them with the message to be sent. The runtime is thus a function of N_S and the message size.

Overall, the client runtime is very efficient. The right figure of Fig. 4 measures the runtime as a function of increased broadcast vector size. In the left figure, since client performance is not affected by the number of non-talking clients (because they do not increase the broadcast vector size), we instead report on the cost of increased trust diversity: how does client runtime change as the number of anytrust servers (and therefor the trust diversity of the system) increase. We find that the costs are indeed far lower than those of Blinder and Riposte

4.3.4 End to end experiments. We evaluate the end-to-end performance of ZIPNet in WAN settings with five and ten servers respectively. The location of servers and the network parameters (latency and bandwidth) between them are shown in Tables 4 and 5. Figure 5 measures the total runtime of a round—from when the aggregator starts processing user submission to the leader server outputs the final broadcast message. Comparing Fig. 5 and Figs. 2 and 3, the total runtime is slightly higher than the sum of the aggregator's and server's processing time obtained from microbenchmarks. This is because the end-to-end runtime includes the time taken for the aggregator to send the final aggregate to all servers.

4.4 Discussion

We now discuss the results of our experiments.

4.4.1 Lower cost of cover traffic. A salient feature of ZIPNet is that cover traffic is cheap. Each additional client message incurs added work by every anytrust server. We cannot do less work for cover traffic messages specifically (this would break anonymity). However, as the number of cover traffic messages increases, we need not increase the size of the broadcast vector. In contrast, for a system that expects more talking clients, we need to use a larger broadcast vector. ZIPNet offers markedly lower costs as the number of clients increases but the broadcast vector size stays the same. As such, cover traffic is cheaper than, e.g., Blinder (see Fig. 6). Moreover, because of message aggregation, servers do not pay $O(B)$ bandwidth per cover message, just a small constant increment due to the client ID (in our implementation a client ID is 32 bytes; with encoding the concrete increment is 84 bytes; see Table 3.).

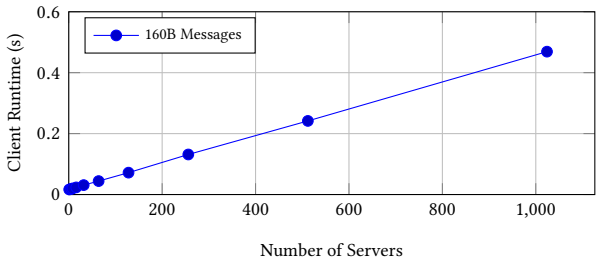
Table 3: Server's bandwidth consumption (bytes) in a round with 1024 *talking clients* and varying numbers of non-talking clients.

Num of clients	Bandwidth	Overhead per non-talking clients
1024	535,607	n/a
2048	622,283	84.64
4000	786,923	84.34
8000	1,123,952	84.26

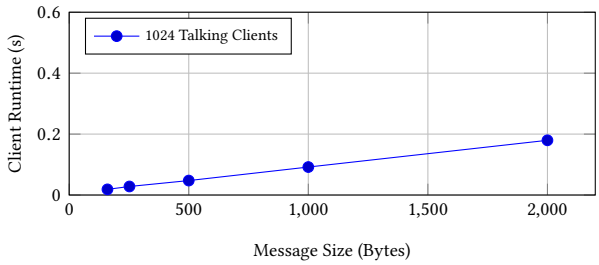
4.4.2 Comparison with other systems. Below we compare ZIPNet with other multi-server anonymous vector schemes. In Section 5, we compare to a wider range of protocols.

Blinder [2]. We empirically compare the server runtime of ZIPNet and CPU Blinder by running them on the same hardware. We obtained Blinder source code from https://github.com/cryptobiu/MPCAnonymousBloging and evaluated its server runtime as the number of clients increased, with 5 and 10 servers respectively. We fixed the message size to 160B and used default parameters in their testing scripts. Figure 6 plots the results with ZIPNet in comparison. We note that we do not compare against GPU Blinder despite it being considerably faster. It is prohibitively costly: Although the minimum resource requirements are unknown, Blinder only reports benchmarks on servers with three server-grade CUDA cards each which cost $24.48 per hour in 2020. This is a >5x cost increase relative to their CPU system for 160-byte messages and 10,000 clients. It is also 65 times the cost of our benchmarked anytrust server, even for 2023 hardware and prices.

With 8000 clients, ZIPNet server is 6.3x and 7.6x faster than CPU Blinder for 5 and 10 servers respectively; with 4000 clients, ZIPNet server is 5.6x and 6.0x faster than CPU Blinder for 5 and 10 servers



(a) Client runtime as the number of servers increases with message size fixed to 160B and 1024 talking users.



(b) Client Runtime as a Function of Message Size.

Figure 4: Client runtime

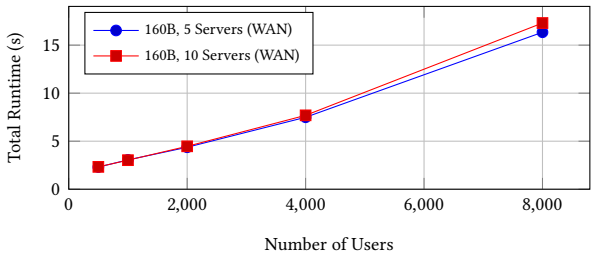


Figure 5: End-to-end round time, from when the aggregator starts processing user requests to the leader server outputs the broadcast message.

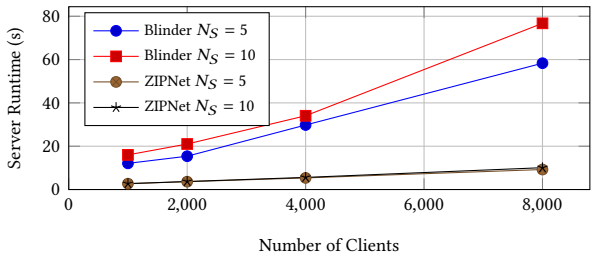


Figure 6: Runtime comparison of ZIPNet's server with CPU Blinder's server running on the same machine (`t2.2xlarge`) in WAN.

respectively. The principal performance improvement in ZIPNet comes from using almost exclusively symmetric cryptography on the server (namely hardware-accelerated AES-NI as a PRF and XOR) in comparison to the asymmetric operations needed by Blinder for its MPC protocols supporting message compression. By outsourcing aggregation to untrusted servers, ZIPNet avoids these costs while achieving lower (near constant) anytrust server bandwidth.

In terms of client runtime, ZIPNet scales much better with the number of servers and message sizes. E.g., with 20 servers, ZIPNet client runtime is far below 0.1s while Blinder takes 40s-75s; even with 5 servers and 1KB message, ZIPNet client runtime is less than 0.2s while Blinder needs 1.4s [2, Fig. 1].

Riposte [18]. ZIPNet's latency is significantly lower than Riposte in the 160B message setting, comparing the 5- and 10-server versions of ZIPNet and the 3-server version of Riposte. CPU Blinder is almost always faster than 3-server Riposte with 5 and 10 servers [2, Fig. 4].

OrgAn [20]. OrgAn presents a novel anonymous broadcast protocol in the client/relay/server setting using almost key-homomorphic PRFs. While anytrust servers are needed for setup, the decryption of messages is done by an untrusted relay, reducing latency.

We compared the end-to-end runtime between ZIPNet and OrgAn under identical hardware configurations. The results show that ZIPNet is orders of magnitude faster: 57x-59x faster with 768 clients, and the advantage increases as the number of clients. In the interest of space, we refer readers to Appendix C for details.

5 RELATED WORK

The dining cryptographer net was introduced by Chaum [16] as a solution to the *dining cryptographer's problem*, wherein a group of cryptographers wishes to establish an anonymous 1-bit broadcast channel for anyone to claim that they paid for dinner (concluding, if 0, that the NSA paid). Chaum's solution, while unconditionally secure, requires every user to communicate with every other user, resulting in a total round communication complexity of $O(BM^2)$, where B is the message size and M is the number of users (recall notation from Table 1).

Many DC nets, including ZIPNet, exist in the *anytrust model*, which trades Chaum's unconditional security for lower communication cost by switching to a client-server communication model. To achieve privacy in the anytrust model, a user need only trust that one of N_S servers is honest. Dissent [19] builds Chaum's system in this model and adds a traitor-tracing protocol to identify users who clobber the output. The servers' bandwidth consumption is linear in the number of users, which limits the scalability to many users.

Riposte [18] improves on Dissent by having stronger trust assumptions and faster disrupter tracing techniques. Riposte's three-server model has two servers that process client requests and a third that audits clients' messages, allowing live detection of disrupters. Since Riposte's topology is the same as that of Dissent other than the addition of a third audit server, it suffers from similar bandwidth limitations. It does improve on this front by using distributed point

functions (DPF), reducing the size of client messages from $O(B)$ to $O(\sqrt{B} + |m|)$.

Blinder [2] in another broadcasting scheme in the anytrust model. It departs from previous works in that it shifts traitor-sensitive behavior to a setup phase of its protocol, whereby each user engages in a secret sharing protocol with the anytrust servers. Assuming the proportion of malicious servers is less than 1/4, malicious users are caught in this phase. After setup, traitor tracing is unnecessary: Blinder relies on secure multiparty computation (MPC) primitives to ensure that malicious users cannot maul round messages. In particular, the anytrust servers engage in a (batched) *format verification* MPC protocol for the secret-shared messages they receive. Blinder uses a similar matrix decomposition trick as Riposte, yielding a user-server bandwidth cost of just $O(\sqrt{B} + |m|)$ per round. The tradeoff for this communication complexity is that the overwhelming majority of server computation time in the CPU version of Blinder is spent on decompressing the plaintext (an MPC operation requiring B multiplications per client message).

Spectrum [32] is an anonymous broadcast system tailored specifically for the case of few, large-payload broadcasters. Its primary improvements lie in its broadcast channel setup and a lightweight traitor tracing protocol. To set up a broadcast in a particular slot, a Spectrum client bootstraps using a less efficient broadcast channel and sends a short *broadcast key* to all servers. After this setup, a client uses knowledge of this broadcast key in order to compute a MAC, permitting it to send in that channel (or send the 0 messages, for cover traffic). Rather than traitor tracing after a round has been mauled, Spectrum does it before: a verifiably encrypted message that fails the *audit* step (a MAC check), is forcibly opened by the servers to determine fault. Using DPFs, Spectrum achieves the user-server bandwidth of $O(\log B + |m|)$ in the case $N_S = 2$, and $O(\sqrt{B} + |m|)$, otherwise.

Express [24] provides enhancements and a change of model. Unlike previous schemes where clients have the potential to miss messages if they do not pay attention to each round, client read and writes are asynchronous. Each message request includes a virtual mailbox, which the owner can read at any point in time. Express improves Riposte's idea of the third audit server by incorporating its functions into the existing two servers, eliminating the need for an additional server. Communication sizes are also reduced from $O(\sqrt{B})$ to $O(1)$ (independent of the number of clients in the system). The limiting constraint of Express is bound by the DPF calculations done on the server.

Clarion [23] provides an elegant MPC shuffling protocol for anonymity approach. The system requires considerable pre-processing overhead for the MPC and this scales with the number of servers and is omitted from their evaluation. They wrote "our evaluation corresponds well to the setting where low latency anonymous broadcast is needed for a short period, e.g., during a live event, that can be planned ahead of time." Clarion outperforms MCMix, [34] an older MPC shuffle system with similar limitations.

OrgAn [20] presents a novel extension to the organizational anonymity setting of PriFi [7], which itself is a version of Dissent tailored to small user bases, such as a campus WiFi network. In PriFi, scheduling blocks on completing an online verifiable shuffle with the any trust servers. To achieve very low latency, this shuffle is not run every round, resulting in somewhat linkable messages. OrgAn replaces the scheduling mechanism with an Almost Key-homomorphic Pseudorandom Function and then employs a classic any-trust DC net for bulk data transmission. While PRF still incurs setup costs with the any-trust servers, the decryption of message can be done by a powerful untrusted party, reducing latency. However, Almost key-homomorphic PRFs have an error term that limits the number of users who can participate in the scheduling round, either as cover traffic or to communicate. Specifically, OrgAn reports benchmarks for at most 200 clients. In contrast, by relying on trusted hardware for scheduling but not privacy, ZIPNet avoids the downsides of both OrgAn and PriFi.

6 CONCLUSION, DEPLOYMENT CONSIDERATIONS, AND ALTERNATIVES TO TEES

ZIPNet is an anonymous broadcast protocol which is 8.7x-17.6x faster than state of the art. By outsourcing message aggregation to untrusted servers, ZIPNet drastically reduces the workload of anytrust servers by removing the need for the server to compute, for every client message, asymmetric cryptographic operations proportional to the size of the broadcast channel. As a result, the cost of additional cover traffic is also minimized. Deploying ZIPNet raises a few considerations.

Churn. ZIPNet is designed to scale to a large number of any trust providers and aggregators. Because aggregators are untrusted for privacy, we envision them being run as infrastructure by a single reputable and competent entity (e.g., like Signal or Tor's directory services). Such a party can split clients over multiple aggregators, preventing a single point of failure.[9]

Failure of a single any-trust server aborts the round, setting a practical limit on how many any-trust servers any scheme can accommodate. We believe ZIPNet gets substantially closer to that limit by reducing operating costs and there are some interesting extensions to decrease the costs of churn are worth further investigation. For instance, a client can participate in a round while only relying on a subset of any-trust servers. Additionally, we can downgrade from any-trust to threshold trust by threshold secret sharing each server's key with the any trust set and operating a consensus mechanism for recovery and failover.

TEEs, SGX, and alternatives. Our prototype uses SGX which Intel recently withdrew from consumer equipment. ZIPNet can be instantiated with any TEE implementation such as keystone [30], Nvidia H100 GPU [33] or ArmTrust Zone. Many of these systems are not yet broadly available or, e.g, for the Secure Elements in iPhones and many Android phones, not yet accessible to developers.

A more interesting question is if we can avoid TEEs completely and rely on software-based methods since we need only correctness, not confidentially. Mobile operating systems provide forms of app attestation (e.g., the DeviceCheck framework in iOS [5]), which may be sufficient. Originally designed to combat ad fraud, these schemes are designed to ensure that requests to a server come only from a genuine instance of an application on a phone.

[9]This would require failover for a root aggregator of aggregators.

Another interesting possibility is whitebox cryptography [39], where it may be possible to deploy multiple different obfuscated versions per client. Because our trust assumptions are falsifiable, the failure of any obfuscated client could be detected and, with an appropriate traitor-tracing scheme, identified and replaced with a fresh obfuscation.

REFERENCES

[1] Rust Crypto. https://github.com/RustCrypto.

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

[3] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder: MPC based scalable and robust anonymous committed broadcast. Cryptology ePrint Archive, Report 2020/248, 2020. https://eprint.iacr.org/2020/248.

[4] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.

[5] Apple Developer Documentation. DeviceCheck. https://developer.apple.com/documentation/devicecheck.

[6] John Barker, Peter Hannay, and Patryk Szewczyk. Using traffic analysis to identify the second generation onion router. In *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*, pages 72–78, 2011.

[7] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Apostolos Pyrgelis, Bryan Ford, Joan Feigenbaum, and Jean-Pierre Hubaux. PriFi: Low-latency anonymity for organizational networks. *PoPETs*, 2020(4):24–47, October 2020.

[8] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014.

[9] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, , and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 441–459, New York, NY, USA, 2017. Association for Computing Machinery.

[10] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. Single secret leader election. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 12–24, 2020.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *Proceedings of the 11th USENIX Conference on Offensive Technologies*, WOOT'17, page 11, USA, 2017. USENIX Association.

[12] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[13] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security Symposium*, 2017.

[14] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to win the clone wars:
efficient periodic n-times anonymous authentication. Cryptology ePrint Archive, Report 2006/454, 2006. https://eprint.iacr.org/2006/454.

[15] David Chaum. Blind signatures for untraceable payments. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *CRYPTO'82*, pages 199–203. Plenum Press, New York, USA, 1982.

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

[17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pages 259–282, 2017.

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

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

[20] Debajyoti Das, Easwar Vivek Mangipudi, and Aniket Kate. OrgAn: Organizational Anonymity with Low Latency. 2022(3):582–605.

[21] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Comprehensive anonymity trilemma: User coordination is not enough. *PoPETs*, 2020(3):356–383, July 2020.

[22] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 21, USA, 2004. USENIX Association.

[23] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. Cryptology ePrint Archive, Report 2021/1514, 2021. https://eprint.iacr.org/2021/1514.

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

[25] FCC. Internet Access Services: Status as of June 30, 2019. https://docs.fcc.gov/public/attachments/DOC-381125A1.pdf, 2022.

[26] The Apache Software Foundation. Teaclave SGX SDK, October 2022. https://github.com/apache/incubator-teaclave-sgx-sdk.

[27] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. ZCash protocol specification version 2022.3.8. Technical report, The ZCash Foundation, 2021.

[28] Engin Kirda and Thomas Ristenpart, editors. *USENIX Security 2017*. USENIX Association, August 2017.

[29] Anna Krasnova, Moritz Neikes, and Peter Schwabe. Footprint scheduling for dining-cryptographer networks. In *International Conference on Financial Cryptography and Data Security*, pages 385–402. Springer, 2016.

[30] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[31] Mixminion. Type III (Mixminion) mix protocol specifications. http://mixminion.net/minion-spec.txt.

[32] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 229–248, 2022.

[33] Nvidia. Confidential computing | NVIDIA. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/.

[34] Anh Pham, Italo Dacosta, Guillaume Endignoux, Juan Ramón Troncoso-Pastoriza, Kévin Huguenin, and Jean-Pierre Hubaux. ORide: A privacy-preserving yet accountable ride-hailing service. In Kirda and Ristenpart [28], pages 1235–1252.

[35] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In Kirda and Ristenpart [28], pages 1199–1216.

[36] Tornado Cash. Core deposit circuit. https://docs.tornado.cash/tornado-cash-classic/circuits/core-deposit-circuit.

[37] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635, 2016. https://eprint.iacr.org/2016/635.

[38] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery.

[39] Brecht Wyseur. White-box cryptography.

A DEFINITION OF BROADCAST ANONYMITY

Here we consider a modified version of the definition proposed in Riposte [18].

A (t, n)-anonymous broadcast system is defined by the following security game between a challenger who operates $t - n$ honest clients and one honest server, and an adversary who operates the remaining parties including an aggregator, the remaining $N_S - 1$ servers, and at most n dishonest clients.

(1) The challenger initializes $t - n$ honest clients, with public keys $PK_c^{\mathcal{H}_1} \dots PK_c^{\mathcal{H}_{t-n}}$, and one honest server with public key $PK_s^{\mathcal{H}_1}$. The honest server's participation threshold is set at $t-n$. The challenger sends these keys to the adversary $\mathcal{A}$.
(2) The adversary picks a static set of corrupt servers and client messages:
    - $\mathcal{A}$ generates N_S corrupt server public keys $PK_s^{\mathcal{A}_1} \dots PK_s^{\mathcal{A}_n}$ and a corrupt aggregator $PK_a^{\mathcal{A}_1}$ public key.

- The adversary specifies the clear text message and the slot for each honest client as $M = \{i, s_i, m_i | i \in [1, t-n]\}$

$\mathcal{A}$ sends $PK_a^{\mathcal{A}_1}$, $PK_s^{\mathcal{A}_1} \dots PK_s^{\mathcal{A}_n}$ and M to the challenger C.

(3) The challenger plays the role of honest clients and the honest server, and allows the adversary to play the role of the aggregator and a malicious network:
   - The challenger registers each honest client with the corrupted servers, the corrupted clients with the honest servers, and the aggregator.
   - The challenger samples a bit b. Let π_0 be the identity permutation and π_1 be a randomly selected permutation over $[1, t-n]$.
   - For each (i, m, s_i) the challenger computes the client's DCnet message c_i for writing m_i to slot s_i under $PK_c^{\mathcal{H}_{\pi_b(i)}}$. That is, depending on b, it either sends the clients messages as specified by the adversary or it swaps which client writes which message according to the permutation.
   - The challenger sends π_1 and $c_1, \dots, c_{(t-n)}$ to the adversary.

(4) The adversary computes a claimed aggregation a of all messages (possibly containing corrupt client messages) which is sent to the challenger.

(5) The challenger computes the output of the honest server $o_{\mathcal{H}}$ given the claimed aggregate a and sends it to the adversary.

(6) The adversary makes a guess b' for the value of b.

The adversary wins the game if $b' = b$.

B ADDITIONAL DETAILS FOR EXPERIMENT SETUP

Table 4: Network bandwidth and latency (RRT in millisecond) of the connections between the aggregator (in `us-east-2c`) and servers.

Server	Avail. Zone	Bandwidth (Mbps)	RTT avg
1	us-east-2b	985	0.84
2	eu-west-3b	127	92.06
3	us-west-1a	246	49.97
4	ap-northeast-1a	84.4	130.96
5	us-west-2c	198	48.48
6	ca-central-1b	522	24.19
7	eu-central-1b	69.6	100.6
8	eu-west-2c	138	85.42
9	sa-east-1c	91.4	124.76
10	us-east-1d	819	11.16

Table 5: Network bandwidth and latency (RRT in millisecond) of the connections between the leader (server #1) and other servers.

Server	Avail. Zone	Bandwidth (Mbps)	RTT avg
2	eu-west-3b	119	91.05
3	us-west-1a	107	51.12
4	ap-northeast-1a	83.7	130.96
5	us-west-2c	84.2	48.3
6	ca-central-1b	66.7	23.49
7	eu-central-1b	234	100.43
8	eu-west-2c	91.1	85.15
9	sa-east-1c	110	124.14
10	us-east-1d	455	10.46

C COMPARISON WITH ORGAN

We acquired the OrgAn source code from https://github.com/zhtluo/organ and assessed its end-to-end runtime as the number of simulated clients increased. Same with above evaluation, we run ZIPNet with aggregator on `c6a.8xlarge` instance and 5 servers on `t2.2xlarge` instances. To ensure similar environment in OrgAn. We used 5 server running on `t2.2xlarge` instance, to generate message representing numerous clients, and a `c6a.8xlarge` functioning as relay. For a consistent comparison, we fixed the message size at 1KB and adhered to the default parameters specified in their testing scripts. Figure 7 illustrates the results of ZIPNet in relation to Organ.

In OrgAn, each round comprises a base phase and a bulk phase. The base phase includes preliminary setups, while the bulk phase involves the majority of data processing, communication, and computation. Our performance comparison between ZIPNet and OrgAn focuses on OrgAn's bulk runtime. OrgAn provides numerous settings including adding blame protocol, unzip protocol and doing delay before sending message. We selected two of the simplest settings, which represent OrgAn's best performance: one without any additional procedures (denoted as rprf) and the other incorporating an unzip operation (denoted as rnoprf).

With 128 clients, ZIPNet demonstrates significant performance enhancements, being 2.2x and 2.4x faster than OrgAn's bulk runtime in settings rprf and rnoprf, respectively. With 768 clients, the efficiency of ZIPNet improves further, exhibiting performance that is 59.1x and 57.6x faster than OrgAn's bulk runtime for the settings rprf and rnoprf, respectively. These results clearly indicate that ZIPNet offers relatively superior performance and scalability compared to OrgAn.

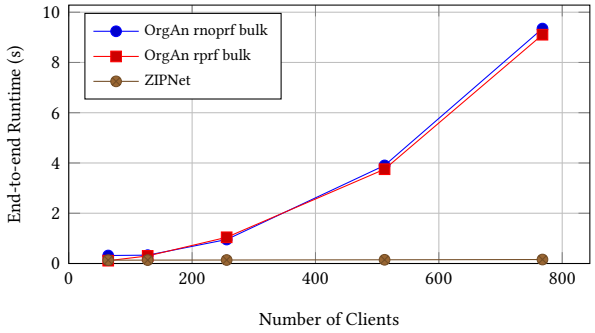


Figure 7: Runtime comparison of ZIPNet and OrgAn on identical hardware in a LAN setting: a `c6a.8xlarge` instance as the aggregator/relay and five `t2.2xlarge` instances as anytrust servers. Both systems were configured with 37 slots, each with 28B per slot.