Atom: Horizontally Scaling Strong Anonymity

Albert Kwon
MIT

Henry Corrigan-Gibbs
Stanford

Srinivas Devadas
MIT

Bryan Ford
EPFL

Abstract

Atom is an anonymous messaging system that protects against traffic-analysis attacks. Unlike many prior systems, each Atom server touches only a small fraction of the total messages routed through the network. As a result, the system's capacity scales near-linearly with the number of servers. At the same time, each Atom user benefits from "best possible" anonymity: a user is anonymous among *all* honest users of the system, even against an active adversary who monitors the entire network, a portion of the system's servers, and any number of malicious users. The architectural ideas behind Atom have been known in theory, but putting them into practice requires new techniques for (1) avoiding heavy general-purpose multi-party computation protocols, (2) defeating active attacks by malicious servers at minimal performance cost, and (3) handling server failure and churn.

Atom is most suitable for sending a large number of short messages, as in a microblogging application or a high-security communication bootstrapping ("dialing") for private messaging systems. We show that, on a heterogeneous network of 1,024 servers, Atom can transit a million Tweet-length messages in 28 minutes. This is over 23× faster than prior systems with similar privacy guarantees.

CCS Concepts

• **Security and privacy → Pseudonymity, anonymity and untraceability**; **Privacy-preserving protocols**; **Distributed systems security**;

Keywords

Anonymous communication, mix-net, verifiable shuffle

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132755

1 Introduction

In response to the widespread electronic surveillance of private communications [69], many Internet users have turned to end-to-end encrypted messaging applications, such as Signal and OTR [2]. These encrypted messaging tools provide an effective way to hide the *content* of users' communications from a network eavesdropper. These systems do little, however, to protect users' *anonymity*. In the context of whistleblowing [38, 76], anonymous microblogging [22], or anonymous surveys [43], users want to protect their identities in addition to the content of their communications.

Unfortunately, anonymity systems that protect against powerful global adversaries typically cannot accommodate large numbers of users. This is primarily due to the fact that traditional anonymity systems only scale *vertically*. These systems consist of a handful of infrastructure servers that act collectively as an anonymity provider; the system can only scale by increasing the power of each participating server. Systems based both on classical mix-nets [18, 48, 72] and on DC-nets [22, 76] suffer from this scalability challenge.

The Tor network [29], in contrast, is an example of an anonymity system that scales *horizontally*. Tor consists of a network of volunteer relays, and increasing the number of these relays increases the overall capacity of the network. This scalability property has enabled Tor to grow to handle hundreds of thousands to millions of users [3]. However, the fact that Tor provides low-latency anonymity also makes the system vulnerable to a variety of deanonymization attacks [9, 17, 31, 44, 56, 74, 75].

In this paper, we present Atom, an anonymous messaging system that takes important steps towards marrying the best aspects of these two architectural strategies. Like Tor, Atom scales horizontally: adding more servers to the network increases the system's overall capacity. Like mix-net- and DC-net-based systems, Atom provides clear security properties under precise assumptions.

Atom implements an *anonymous broadcast primitive* for short, latency-tolerant messages. In doing so, Atom offers a strong notion of anonymity: an adversary who monitors the entire network, a constant fraction of servers, and any number of users only has a negligible advantage at guessing which honest user sent which message.

We target two applications in particular in this paper. The first is an *anonymous microblogging* application. With Atom, users can broadcast short messages anonymously to organize protests, whistleblow, or send other sensitive messages.

The second is a "*dialing*" application: many existing private messaging systems [8, 50, 72] require pairs of users to first establish shared secrets using some out-of-band means. Atom can implement this sort of dialing system while providing strictly stronger security properties than prior schemes can.

An Atom deployment consists of hundreds or thousands of volunteer servers, organized into small groups. To use the system, each user submits its encrypted message to a randomly chosen entry group. Once each server group has collected ciphertexts from a number of users, the group shuffles its batch of ciphertexts, and forwards a part of each batch to neighboring server groups. After the servers repeat this shuffle-and-forward process for a certain number of iterations, our analysis guarantees that no coalition of adversarial servers can learn which user submitted which ciphertext. At this point, each group decrypts the ciphertexts it holds to reveal the anonymized plaintext messages.

Atom's scalability comes from the fact that each group of servers works *locally*, and only needs to handle a small fraction of the total messages routed through the network. In an Atom deployment routing M messages using N servers, each Atom server processes a number of ciphertexts that grows as $\tilde{O}(M/N)$. In contrast, traditional verifiable-shuffle-based or DC-net-based anonymity systems require each server to do $\Omega(M^2)$ work, irrespective of the number of servers in the system [22, 48, 76].

The design of Atom required overcoming three technical hurdles. First, in a conventional mix-net, each user produces an onion-style ciphertext, in which her message is encrypted to each of the mix servers. In Atom, the user does not know the set of servers its message will travel through a priori, so she does not know which servers' keys to use to encrypt her message. Prior designs for distributed mix systems [55, 63, 78] circumvented this problem with general-purpose multi-party computation (MPC) protocols [11, 35], but these general methods are currently too inefficient to implement. We instead use a new rerandomizable variant of ElGamal [32] encryption, which allows groups of servers in the network to collaboratively and securely decrypt and reencrypt a batch of ciphertexts to a subsequent group.

Second, Atom must maintain its security properties against *actively* malicious servers. To protect against active attacks, we group the servers in such a way that every group contains at least one honest server with overwhelming probability. We then rely on the honest server to ensure that certain invariants hold throughout the system's execution using two different cryptographic techniques. The first method relies on verifiable shuffles [33, 39, 59], which can proactively identify bad actors but is computationally expensive. The second method is a novel "trap"-based scheme, inspired by prior work on robust mixing [46]. This scheme avoids using expensive verifiable shuffles, but provides a slightly weaker notion of security: a malicious server can remove κ honest users from the system (without deanonymizing them) with probability $2^{-\kappa}$, thereby reducing the size of the remaining users' anonymity set.

Finally, Atom must handle network churn: with hundreds or thousands of servers involved in the network, benign server failures will be common. Our mechanism allows us to pick a fault-tolerance parameter h, and Atom can tolerate up to $h-1$ faults per group while adding less than two seconds of overhead. Atom also provides a mechanism for recovering from more than $h-1$ faults with some additional overhead.

To evaluate Atom, we implemented an Atom prototype in Go, and tested it on a network of 1,024 Amazon EC2 machines. Our results show that Atom can support more than one million users sending microblogging messages with 28 minutes of latency using 1,024 commodity machines. (Processing this number of messages using Riposte [22], a centralized anonymous microblogging system, would take more than 11 hours.) For a dialing application, Atom can support a million users with 28 minutes of latency, with stronger guarantees than prior systems [50, 72].

In this paper, we make the following contributions:

- propose a horizontally scalable anonymity system that can also defend against powerful adversaries,
- design two defenses to protect this architecture against active attacks by malicious servers,
- design a fault-recovery mechanism for Atom, and
- implement an Atom prototype and evaluate it on a network of 1,024 commodity machines.

With this work, we take a significant step toward bridging the gap between scalable anonymity systems that suffer from traffic-analysis attacks, and centralized anonymity systems that fail to scale.

2 System overview

Atom operates by breaking the set of servers into many small groups such that there exists at least one honest server per group with high probability; we call such group an *anytrust* [76] group. We then connect the groups using a carefully chosen link topology.

Communication in Atom proceeds in time epochs, or protocol *rounds*. At the start of each round, every participating user holds a plaintext message that she wants to send anonymously. To send a message through Atom, each user pads her message up to a fixed length, encrypts the message, and submits the ciphertext to a user-chosen *entry group*. Each entry group collects a predetermined number of user ciphertexts before processing them.

After the collection, each group collectively shuffles their set of messages. The group collaboratively splits the shuffled

messages into several batches and forwards each batch to a different subsequent group. This shuffle-split-and-forward procedure continues for a number of iterations, until a set of *exit groups* finally reveal the users' anonymized messages. The exit servers can then forward these messages to either a public bulletin board (e.g., for microblogging) or an address specified in the message (e.g., for dialing).

Converting this high-level architecture into a working system design requires solving a number of challenges:

(1) How does Atom provide protection against traffic-analysis attacks by a global adversary? (§3)
(2) To whom do clients encrypt their messages? If clients do not know which servers their messages will pass through, standard mix-net-style onion encryption will not suffice. (§4.2)
(3) How does Atom protect users from malicious servers who deviate from the protocol? (§4.3 and §4.4)
(4) How does Atom remain resilient against server churn? (§4.5)

2.1 Threat model and assumptions

An Atom deployment consists of a distributed network of hundreds or thousands of servers, controlled by different individuals and organizations. A cryptographic public key defines the identity of each server, and we assume that every participant in the system agrees on the set of participating servers and their keys. (A fault-tolerant cluster of "directory authorities" could maintain this list, as in the Tor network [29].) Furthermore, we assume that the servers and the clients communicate over encrypted, authenticated, and replay-protected channels (e.g., TLS). A large number of users—on the order of millions—can participate in each round of the Atom protocol.

We assume that the adversary monitors all traffic on the network, controls a constant fraction f of the servers, and can control all but two of the users. The adversarial servers may deviate from the protocol in an arbitrary way, and collude with each other and adversarial users.

Atom can provide availability in the presence of fail-stop server faults (§4.5), but not against Byzantine server faults [49]. Our fault-tolerance technique could mitigate some availability attacks, but we leave availability attacks out of scope. That said, Atom *does* protect users' anonymity in all cases.

Finally, Atom does not attempt to prevent intersection attacks [28, 45]. For example, if anonymous messages about a protest in Turkey are only available when Alice is online, then the adversary may be able to infer that Alice is sending those messages. Atom does not protect against this attack, but known techniques [41, 77] can mitigate its effectiveness.

2.2 System goals

Atom has three primary goals.

Correctness. At the end of a successful run of the Atom protocol (i.e., a run that does not abort), each server holds a subset of the plaintext messages sent through the system. Informally, we say that the system is *correct* if the union of the message sets held at all honest servers contains every message that the honest users sent through the system.

Anonymity. Following prior work [16, 18, 22, 48, 76], we say that Atom provides *anonymity* if an adversary who controls the network, a constant fraction of the servers, and any number of users cannot guess which honest user sent which message with probability non-negligibly better than random guessing. In particular, we require that the final permutation of the honest users' messages is indistinguishable from a random permutation. Under this definition, a user is anonymous among *all* honest users, not just the users who share the same entry group, even if there is only one honest user in an entry group.

Scalability. We say that an anonymity system is *scalable* if the system can handle more users as the number of servers grows. If there are M messages and N servers in the system, we denote the number of ciphertexts each server processes as $\mathcal{C}(M, N)$. We then require that holding $\mathcal{C}(M, N)$ fixed, M scales linearly with N.

2.3 Cryptographic primitives

Atom relies on the following two cryptographic primitives. We describe them at a high level here, and give the details in Appendix A.

Rerandomizable encryption. Atom uses a rerandomizable CPA-secure encryption scheme, which consists of the following algorithms:

- $(sk, pk) \leftarrow \mathsf{KeyGen}()$. Generate a fresh keypair.
- $c \leftarrow \mathsf{Enc}(pk, m)$. Encrypt message m using public key pk.
- $m \leftarrow \mathsf{Dec}(sk, c)$. Decrypt ciphertext c using secret key sk.
- $C' \leftarrow \mathsf{Shuffle}(pk, C)$. Rerandomize a vector C of ciphertexts using public key pk, and randomly permute the components of the vector.
- $c' = \mathsf{ReEnc}(sk, pk, c)$. Strip a layer of encryption off of ciphertext c using secret key sk and add a layer of encryption using public key pk. When $pk = \perp$, this operation is the same as $\mathsf{Dec}(sk, c)$.

For Atom, we require an additional property that the cryptosystem allow "out-of-order" decryption and reencryption. That is, given an onion-encrypted ciphertext, a server can decrypt one of the middle layers of encryption and reencrypt the ciphertext to a different public key.

Non-interactive zero-knowledge proofs of knowledge (NIZKs). We make use of three NIZK constructions. We use

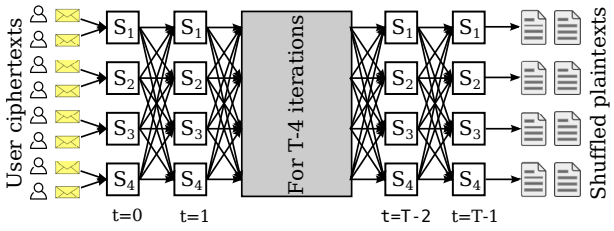

Figure 1. An Atom deployment with four servers (S_1 through S_4), arranged in the square topology. Each user submits her ciphertext to an entry server of her choice. The servers perform T iterations of mixing before outputting the plaintexts.

non-malleable NIZKs, meaning that the adversary cannot use a NIZK for an input to generate a different NIZK for a related input.

- $(c, \pi) \leftarrow \mathsf{EncProof}(pk, m)$. Compute $c \leftarrow \mathsf{Enc}(pk, m)$ and generate a NIZK proof of knowledge of the plaintext m corresponding to c.
- $(c, \pi) \leftarrow \mathsf{ReEncProof}(sk, pk, m)$. Compute $c \leftarrow \mathsf{ReEnc}(sk, pk, m)$ and generate a NIZK proof π that this operation was done correctly (cf. [20]).
- $(C', \pi) \leftarrow \mathsf{ShufProof}(pk, C)$. Permute the ciphertext set C (using Shuffle) and generate a NIZK π that C is a permuted version of C', reblinded using pk (cf. [10, 39, 59]).

When the operations other than Shuffle and ShufProof are applied to a vector of ciphertexts C, we apply the operation to each component of the vector.

3 Random permutation networks

In this section, we describe our solution to the first challenge: how does Atom provide protection against traffic-analysis attacks by a global adversary? We use a special network topology that can provide anonymity against an adversary that can view the entire network and control any number of system users but that *controls no servers*. Then, in §4, we describe how to protect against malicious servers.

We organize the Atom servers into a layered graph, and each Atom server is connected to β other servers in the next layer, where β is a fixed branching factor. An example topology is shown in Figure 1 for $\beta = 4$. In each protocol run, each user first chooses an Atom server (an "entry server") and encrypts her message with the public key of the server using a rerandomizable encryption scheme. Each user then sends her ciphertext to the entry server. Along with her ciphertext, the user also provides a NIZK to prove she knows the underlying plaintext, to prevent a malicious user from submitting a rerandomized copy of an honest user's ciphertext; this would result in duplicate plaintexts at the end of the network, which would immediately reveal the sender of the message. We include the id of the server in the NIZK generation to prevent a malicious user from resubmitting the exact copy of a ciphertext and its NIZK to a different server. The details of this NIZK are shown in Appendix A.

We assume that each entry server receives the same number of messages. To achieve this in practice, an untrusted load-balancing server could direct users to different entry servers. We argue later in this section that every honest user of the system is anonymous amongst *all* honest users of the system. Thus, a user's choice of entry group is not important for security. Moreover, the untrusted server just directs the users to a server, and does not actually route any messages. It therefore cannot selectively remove users' messages in an attempt to launch an intersection attack.

Once the Atom entry servers collect a certain pre-specified number of ciphertexts, the servers then begin a mixing process that repeats for T iterations, where T is a parameter chosen later on. In each iteration of the mixing process, each server performs the following steps:

- Randomly permute the set of ciphertexts.
- Divide the ciphertexts into β batches of equal size.
- Reencrypt batch $i \in [1, \beta]$ for the i^{th} neighboring server and forward the reencrypted batch to that server.

The servers repeat this permute-and-reencrypt process using the incoming batches. After T iterations, each server in the network holds a set of ciphertexts that has passed through T other servers in the network. The *exit servers* can then decrypt these ciphertexts and publish the corresponding plaintexts.

Security analysis. We use a particular network topology called a *random permutation network* [23–26, 40] to connect the servers. When the servers are organized into such a network, the output of the network after carrying out the protocol above is a near-uniform random permutation of all input messages. The adversary will thus have negligible advantage in guessing which honest user sent which message over random guessing. Atom is compatible with any good random permutation network. In this work, we leverage prior analyses to identify two simple candidate topologies:

Square network: Håstad studied the problem of permuting a square matrix of M elements by repeatedly permuting the rows and columns [40]. His analysis gives rise to a random permutation network on $\sqrt{M}$ nodes in which each vertex shuffles $\sqrt{M}$ ciphertexts and connects to $\sqrt{M}$ vertices on the subsequent layer (Figure 1). Håstad demonstrated that this network produces a near-uniform random permutation after only $T \in O(1)$ iterations of mixing. In general, when we have many more messages than servers (in particular when $N < \sqrt{M}$), we can have each server "be responsible" for multiple vertices in the network. For example, S_1 and S_2 in $t = 0$ and 1 in Figure 1 could be handled by a single server.

Iterated-butterfly network: It was shown that $O(\log M)$ repetitions of a standard butterfly network yields an almost-ideal random permutation network on M elements [26], where each node in the butterfly network handles $O(1)$ messages. We say "almost ideal" because the network produces a random permutation on a constant fraction of the M elements. Adding a small constant fraction of dummy messages to the system lets us use this network as if it produced a truly random permutation [26]. Since a butterfly network has depth $O(\log M)$, the total depth of the network is $O(\log^2 M)$. If there are fewer than M servers, then we again have a single server emulate multiple nodes. The resulting N-server network topology would have $O(\log^2 N)$ depth, meaning there would be $T \in O(\log^2 N)$ iterations of mixing.

Efficiency. With M total messages in the network, each server in each mixing iteration handles M/N messages. For the square network, the number of mixing iterations satisfies $T \in O(1)$, and thus each server only handles $O(M/N)$ ciphertexts total in the square network. In the case of the iterated-butterfly network, the number of messages each server handles is $O(\frac{M}{N} \cdot \log^2 N)$, since there are $T \in O(\log^2 N)$ iterations. For the rest of this paper, we focus on the square network, since it will perform better in practice due to the shallower depth. Both networks meet the scalability requirement as both can handle more messages as we increase the number of servers with a fixed $C(M, N)$.

4 Atom protocol

In this section, we extend the protocol of §3 to defend against actively malicious servers (i.e., servers who can tamper with the messages). At a high level, we divide the servers into anytrust groups, and replace the servers in the network with the groups. Each group then simulates an honest server using our protocol, and we provide mechanisms to detect malicious servers deviating from the protocol.

4.1 Anytrust group formation

The Atom servers are organized into groups. As we will see later in this section, the security of Atom relies on each group having at least one honest server in this setting. We ensure this by using a public unbiased randomness source [14, 68] to generate groups consisting of randomly sampled servers. We set the group size large enough to ensure that the probability that all servers in any group are malicious is negligible. Here, we make an assumption that the adversary controls at most a particular constant fraction f of the servers in the network.

For example, if we assume that the adversary controls $f = 20\%$ of the servers, we can bound the probability that any group consists of all malicious servers ("is bad") by choosing the group size k large enough. We compute:

$$\Pr[\text{One group of } k \text{ servers is bad}] \leq f^k,$$

Algorithm 1 Basic Atom group protocol

Form anytrust groups using the protocol in §4.1. A group then takes a set of ciphertexts C as input, either from the users if this server is in the first layer of the network or the groups in the prior layer, and executes the following.

(1) **Shuffle**: Each server s computes $C' \leftarrow \mathsf{Shuffle}(C)$ in order, and sends C' to the next server.
(2) **Divide**: If s is the last server of the group, then it divides the permuted ciphertext set C' into β evenly sized batches $(B_1, \ldots, B_\beta)$, where β is the number of neighboring groups. It sends $(B_1, \ldots, B_\beta)$ to the first server of the current group.
(3) **Decrypt and Reencrypt**: Server s receives batches $(B_1, \ldots, B_\beta)$ from the previous server. In order, each server s computes $B'_i = \mathsf{ReEnc}(sk_s, pk_i, B_i)$ for each batch where sk_s is the secret key of the current server and pk_i is the group public key of the ith neighboring group. If there are no neighbors (i.e., this is the last iteration of mixing), then $pk_i = \perp$ for all i.
If s is the last server of the group, then it sends all B'_i to the first server in the i^{th} neighboring group. Otherwise, it sends them to the next server.

If s is the last server of a group in the last layer, then $(B'_1, \ldots, B'_\beta)$ contains the plaintext messages.

and then use the union bound to compute

$$\Pr[\text{Any of } G \text{ groups of } k \text{ servers is bad}] \leq G \cdot f^k.$$

If we allow failure with probability at most 2^{-64} with $G = 1024$ groups, then we choose the group size $k = 32$ such that $f^k \cdot G < 2^{-64}$. Finally, we replace the servers with the groups in the permutation network.

New groups are formed at the beginning of each round. In practice, this operation will happen in the background, as the rest of the protocol is carried out.

4.2 Basic Atom protocol

We first describe our protocol that enables each group to collectively shuffle and reencrypt the message, without protection against active attacks. To send message m, the user first picks her entry group. She then computes $(c, \pi) \leftarrow \mathsf{EncProof}(pk, m)$, where pk is the *group public key*; this produces an onion-encrypted ciphertext encrypted using the public keys of all servers in the group. For ElGamal, for example, pk would be the product of the public keys of all servers in the group. She then sends (c, π) to all servers in the entry group, and π is verified by all servers.

Once enough ciphertexts and proofs are received, each group performs a specialized multi-party protocol for shuffling the messages described in Algorithm 1. The security of this protocol relies on Step 1 and Step 3. After Step 1, the

messages are permuted using the composition of all servers' permutations. Since the honest server's permutation is random and unknown, the final permutation is secure as well.

Step 3 solves our second challenge: to whom do the users encrypt their messages? Here, we use the special out-of-order reencryption property of our cryptosystem (§2.3 and Appendix A). A user only needs to encrypt for her entry group (and need not know the path her message will take a priori), and the entry group can reencrypt for the appropriate next group. In particular, the first server decrypts a layer of encryption of a ciphertext and reencrypts it for the next group of servers. When the second server receives the resulting ciphertext, it can decrypt its layer of encryption, despite the fact that the ciphertext was last encrypted with a different public key. Since each message is simultaneously decrypted and reencrypted for another group, all messages remain encrypted under at least one honest server's key until the last layer. Thus, the adversary does not learn anything by observing the traffic in intermediate mixing iterations.

4.3 Atom with NIZKs

We now describe the two different mechanisms that address our third challenge: protecting against actively malicious servers. First, each user generates her ciphertext and the corresponding NIZK using EncProof, and submits them to all servers in their entry groups. All servers then verify the NIZKs, and report the verification result to all other servers in the group. Our protocol then uses *verifiable shuffles* [10, 39, 59] and *verifiable decryption* [20]: After each operation in Algorithm 1, the server who shuffled or reencrypted the messages proves the correctness of its operation using NIZKs to all other servers in the group. The honest server in each group will detect any deviation from the protocol. Algorithm 2 describes the details.

4.4 Atom with trap messages

Generating and verifying the zero-knowledge proofs imposes a substantial computational cost on the servers. The verifiable shuffle proposed by Neff [59], for instance, requires each server to perform a number of exponentiations per element being shuffled. Since every server needs to produce and verify a number of these NIZK proofs, the extra computation can be burdensome. Instead, we use a novel *trap message*-based protection, inspired by prior work [46]. In this variant, each user submits a "trap" ciphertext with the ciphertext of her message. If a server misbehaves, it risks tampering with a trap. We then provide a distributed mechanism to detect if a trap has been modified.

The malicious servers may get lucky and tamper with a real user message. As such, the trap variant of Atom provides a slightly weaker notion of security than the NIZK variant: the adversary could remove (but not deanonymize) up to κ honest messages from the anonymity set with probability $2^{-\kappa}$. If the number of honest users is large, as we expect in a scalable system, this weaker anonymity property is almost as good as the traditional anonymity property. The NIZK variant of Atom can provide the stronger anonymity if needed.

This trap variant of Atom makes use of an extra anytrust group of servers, which we call the *trustees*. The trustees first collectively generate a per-round public key for the group, with each trustee holding a share of the matching secret key. Users then encrypt their messages using a double-enveloping technique [37]: each user first encrypts her messages using

Algorithm 2 Atom group protocol with NIZKs

A group receives a set of ciphertexts C as input, as well as the proof verification results of the previous layer.

(1) **Shuffle**: In order, each server s does the following:
   (a) Compute $(C', \pi) \leftarrow \mathsf{ShufProof}(pk, C)$, where pk is the current group public key.
   (b) Send (C', π) to all servers in the group. All servers in the group verify the proof π, and send the result of the verification to all servers in the group. All servers then check that every server in the group correctly verified the proof, and abort the protocol if any server reports failure.

(2) **Divide**: If s is the last server in the group, then server s divides the permuted ciphertext set C' into β evenly sized batches $(B_1, \ldots, B_\beta)$, where β is the number of neighboring groups. Server s sends $(B_1, \ldots, B_\beta)$ to the first server.

(3) **Decrypt and Reencrypt**: In order, each server s does the following after it receives batches $(B_1, \ldots, B_\beta)$ from the previous server:
   (a) Compute $(B'_i, \pi_i) = \mathsf{ReEncProof}(sk_s, pk_i, B_i)$ for each batch B_i, where sk_s is the secret key of the current server and pk_i is the group public key of the i^{th} neighboring group.
   (b) Send $\{(B'_i, \pi_i)\}_{i \in [\beta]}$ to all servers in the current group. Send them to all servers in all the neighboring groups as well, if s is the last server of the group. All servers that received the proofs $\{\pi_i\}_{i \in [\beta]}$ verify them, and send the results of the verification to all servers in the current group, and the neighboring groups if s is the last server in the group. Then, all servers who received the proofs check that all other servers who verified the proofs reported success, and abort the protocol if any server reports failure.

If s is the last server of a group in the last layer, then $(B'_1, \ldots, B'_\beta)$ contains the traps and the inner ciphertexts.

the trustees' public key with an IND-CCA2 secure encryption scheme [58, 62], which ensures that the resulting ciphertext cannot be modified in anyway. Then, the user encrypts the resulting ciphertext, which we call *inner ciphertexts*, with the group key of her entry group. More precisely, to send a message m, a user

(1) encrypts m using the trustees' public key pk_T: $c_M \leftarrow$ "$\mathsf{Enc}_{CCA2}(pk_T, m)\|\mathtt{M}$", where M indicates that c_M is an inner ciphertext,
(2) picks an entry group. Let *gid* be the index of the entry group,
(3) chooses a random nonce R and generates a "trap message" as $c_T \leftarrow$ "$gid\|R\|\mathtt{T}$", where T indicates that c_T is a trap message,
(4) computes $(c_0, \pi_0) \leftarrow \mathsf{EncProof}(pk, c_M)$, $(c_1, \pi_1) \leftarrow \mathsf{EncProof}(pk, c_T)$, and the cryptographic commitment C_T of c_T, where pk is the public key of the entry group (since the nonces are high-entropy, we can use a cryptographic hash like SHA-3 [30] as a commitment),
(5) sends (c_0, π_0) and (c_1, π_1) in a random order along with C_T to all servers in her entry group.

The servers verify π_0 and π_1 when they receive them. Once a group collects enough ciphertexts and commitments, each group carries out Algorithm 1, treating each ciphertext (real or trap) as an independent message. At the end of the protocol, the last server checks its subset of messages. It then forwards (1) each trap message to all servers in the group indicated by the *gid* field, and (2) each inner ciphertext to all servers in the group chosen by a deterministic function that will load-balance the number of ciphertexts forwarded to a group (e.g., using universal hashing).

Each server reports the following to the trustees:

(1) a bit indicating whether every trap commitment has a matching trap and vice-versa.
(2) a bit indicating that all inner ciphertext has been forwarded correctly (e.g., hash is the expected value), and that there are no duplicate inner ciphertexts.
(3) the number of traps and inner ciphertexts.

Each trustee releases its share of the decryption key if and only if every server in every group reports no violation and the total number of traps is the same as the total number of inner ciphertexts. Otherwise, each trustee deletes its share of the secret key. If the trustees release the decryption key, then each server decrypts the inner ciphertexts to recover the actual messages. Figure 2 summarizes the protocol.

Security analysis. The messages sent in one round cannot be replayed in another round because the group keys change across rounds. Thus, we only consider the security of one round. The servers cannot tamper with any trap messages since the honest server in each group holds the commitments

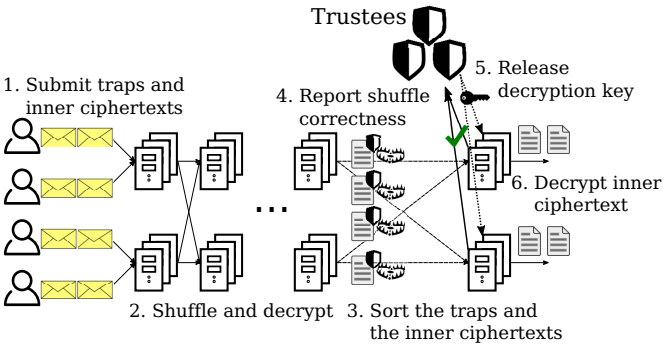

Figure 2. Atom with trap messages and trustees.

to all traps it is expecting to see. Moreover, because the traps are mixed independently of the real messages, the final locations of traps do not leak any information about the real messages. We then use IND-CCA2 encryption, which creates non-malleable ciphertexts, to prevent the adversary from tampering with the inner ciphertexts to create related ciphertexts. The adversary could still, however, duplicate, drop, or replace an inner ciphertext.

The servers first explicitly check for duplicates, and abort the protocol upon finding any. When a malicious server removes or replaces a ciphertext, there is at least 50% chance that the modified ciphertext is a trap message because the users submit the ciphertexts in a random order and the ciphertexts are indistinguishable. Thus, if a server drops or replaces a single ciphertext, it causes the entire protocol run to abort with probability 50%. The adversary can, however, remove or replace κ messages successfully with probability $2^{-\kappa}$. Since each successful tampering reduces the anonymity set of all users by one, the adversary can reduce the anonymity set size by at most κ with probability $2^{-\kappa}$. This attack does not impact the privacy of the tampered ciphertexts: the removed inner ciphertexts are always encrypted under at least one honest server's key, and thus the plaintext messages of the replaced messages are never revealed.

4.5 Tolerating server churn

The failure of any server–even a benign one–in the protocols described thus far prevents the failed server's group from making progress. This is an important challenge for Atom: with hundreds or thousands of volunteer servers involved in each round, server failures are bound to happen. To address this issue, we modify Atom to use threshold anytrust groups that we call "many-trust groups". We construct the groups such that there are at least h honest servers in each group, and enable each group to tolerate up to $h - 1$ failures.

When using Atom with many-trust groups, we replace each group public key with the key of a threshold cryptosystem. In a k-server group, we share the keys in such a way

that any $k-(h-1)$ servers can decrypt messages encrypted for the group's key. Since there are at least h honest servers in each group, any subset of $k-(h-1)$ servers contains an honest server. With such groups, Atom works similar to the anytrust variant, except that only $k-(h-1)$ group members need to participate.

Our fault-tolerance mechanism requires two changes to the network setup described in §4.1. First, we need to increase the group size k to ensure existence of h honest servers per group. For example, when $h=2$, $f=20\%$, we need $k \geq 33$ to achieve failure probability $< 2^{-64}$ (compared to $k \geq 32$ when $h=1$). In the common case, however, only $32 = k-(h-1)$ servers need to handle the messages, meaning the messaging latency does not increase in this case. Appendix B shows how to compute k, and how large k must be for different values of h.

Second, each group must generate its threshold encryption keys. In Atom, we use a dealer-less distributed verifiable secret sharing (DVSS) protocol [67] to generate the keys to avoid having a single trusted dealer who knows the secret key. When a group is first formed, all servers in the group participate in a round of the DVSS protocol. The public output of this protocol is the group public key and the secret key is secret shared among the k servers. For subsequent rounds, the servers can perform this operation in the background as they mix the messages for the current round.

Atom also provides a way to recover from more than $h-1$ failures in a group using what we call *buddy groups*. When groups are formed in the beginning of a round, each group picks one or more buddy groups. Each server then secret shares its share of the group private key with the servers in each of the buddy groups. When more than $h-1$ servers in a group fail, a new anytrust group is formed. Each server in the new group then collects the shares of the private key from one of the buddy groups, and reconstructs a share of the group private key. This allows Atom to recover from a group failure as long as one of its buddy groups is online. In principle, Atom could use an extra anytrust group of highly available servers not in the actual network as a buddy group for all groups to minimize the chance of network failure; for example, the trustee group can be used for this purpose in the trap-variant of Atom.

4.6 Malicious users in Atom

In the NIZK variant of Atom, malicious users cannot cause a protocol run to halt. In the trap variant, however, malicious users could potentially disrupt a round by submitting (1) missing, incorrect, or extra traps, or (2) duplicate inner ciphertexts. Since the servers check the traps only once the routing has completed, Atom with traps unfortunately cannot proactively prevent such attacks. However, Atom does provide a way to identify the malicious users after a misbehavior is detected. To identify disruptive users, all entry groups first reveal their private keys. Then, all servers in each group decrypt the ciphertexts they received, and publish the resulting traps and inner ciphertexts to all other servers, along with the original sender of each message and the commitments of the traps. Each server then checks if each decrypted trap matches the corresponding commitment and vice-versa, and reports any user who fails this check. It also reports any users who submitted the same inner ciphertexts. Once the malicious users' identities are known, the system maintainer could take appropriate actions (e.g., blacklist the users). If the DoS attack is persistent after many rounds, Atom can fall back to using NIZKs, effectively trading off performance for availability.

4.7 Organizing servers

Ensuring maximal server utilization. To maximize the performance, we need to fully utilize all servers at all times. A naïve layout of the servers, however, will cause a lot of idle time. For example, the second server in a group cannot do any useful work until the first server finishes. To ensure that every server is active as much as possible, we "stagger" the position of a server when it appears in different groups (e.g., server s is the first server in the first group, second server in the second group, etc.). This can help minimize idle time. Changing the positions of the servers within a group does not impact the security of the system, since the security only depends on the existence of an honest server.

Pipelining. In scenarios where throughput is more important than latency, Atom can be pipelined. When we organize the servers, we can assign different sets of servers to different layers of our network. The network can then be pipelined layer by layer, and output messages every one group's worth of latency. We do not explore this trade-off in this paper, as latency is more important for the applications we consider.

5 Implementation and applications

We implemented an Atom prototype in Go in approximately 3,500 lines of code, using the Advanced Crypto Library [1]. Our prototype implements both protection against active attacks (§4.3 and §4.4) and our fault-tolerance protocol (§4.5). We use the NIST P-256 elliptic curve [6] for our cryptographic group, threshold ElGamal encryption [65] for our fault-tolerance scheme, and Neff's verifiable shuffle technique for the zero-knowledge proof of shuffle correctness [59]. For our IND-CCA2-secure encryption scheme, we use a key encapsulation scheme with ElGamal [66], described in Appendix A. The source code is available at github.com/kwonalbert/atom. We now highlight two particularly suitable applications for Atom.

Microblogging. Microblogging is a broadcast medium in which users send short messages. For example, Twitter is a microblogging service that supports messages up to 140 characters. In several cases, anonymity is a desirable property for microblogging: protest organizers can announce their plans, and whistleblowers can publicly expose illegal activities without fearing repercussions. Atom is a natural fit for such applications. To blog, a user sends a short message through the Atom network. The servers then put the plaintext messages on a public bulletin board where other users can read them. We use 160 byte messages in our evaluation.

Dialing. Several private communication systems [8, 50, 72] require a *dialing* protocol by which pairs of users can establish a shared secret. Atom supports a dialing protocol, similar to that of Vuvuzela [72] and Alpenhorn [50]. To dial another user Bob, the initiator Alice encrypts her public key using Bob's public key. Then, she sends her encrypted key and Bob's identifier *id* (e.g., his public key) through the Atom network. The servers at the last layer put the encrypted keys into *mailboxes* based on the identifier. There are m mailboxes, and each dialing message is forwarded to mailbox $id \bmod m$. Finally, Bob downloads the contents of the mailbox that contains the requests for him, decrypt the messages, and establishes a shared key with Alice.

To hide the number of dialing calls a user receives, we employ the differential privacy technique proposed by Vuvuzela. We require one of the anytrust groups (which could be the trustees in the trap variant) to generate dummy dialing messages for each mailbox, where the number of dummies is determined using differential privacy. Then, these dummy messages are distributed evenly across the Atom network, and routed along with the actual users' dialing messages. We refer the readers to prior work [72] for detailed privacy analysis of the required number of dummies.

The size of each dialing message can differ depending on the cryptosystem used and the security guarantees. In the simplest version in which a user simply sends an encrypted public key of an elliptic curve cryptosystem, the message size can be as small as 80 bytes (Bob's public key + AEAD encrypted Alice's public key). For more sophisticated dialing protocols, the messages can be larger. Alpenhorn [50], for instance, uses identity-based encryption, and each message is approximately 300 bytes. Our prototype implements the simpler 80 byte message dialing scheme.

6 Evaluation

To evaluate Atom, we performed two classes of experiments. In the first set of experiments, we measured the performance of a single anytrust group. In the second, we perform end-to-end experiments using a large number of machines. In these latter experiments, we verify that the system can (1) handle a large number of messages, and (2) scale horizontally. We carried out our experiments in the us-east-1 region of Amazon EC2, but we artificially introduced a latency between 40 and 160 ms for each pair of servers (using `tc` in Linux) to emulate a more realistic network environment. The machines communicated over TLS channels.

Table 3. Performance of the cryptographic primitives.

Primitives	Latency (s)	
Enc	$1.40 \cdot 10^{-4}$	
ReEnc	$3.35 \cdot 10^{-4}$	
Shuffle (1,024 messages)	$1.07 \cdot 10^{-1}$	
	Prove	Verify
EncProof	$1.62 \cdot 10^{-4}$	$1.39 \cdot 10^{-4}$
ReEncProof	$6.55 \cdot 10^{-4}$	$4.46 \cdot 10^{-4}$
ShufProof (1,024 messages)	$7.57 \cdot 10^{-1}$	$1.41 \cdot 10^{0}$

Table 4. Latency to create an anytrust group.

Group size	4	8	16	32	64
Setup latency (ms)	7.4	29.4	93.3	361.8	1432.1

6.1 Anytrust group performance

Experimental setup. To understand the performance of the building blocks of an Atom network, we measured the latency of the cryptographic operations and one mixing iteration. We used Amazon EC2's c4.xlarge instances for most of the experiments in this section, which have four Intel Haswell Xeon E5-2666 vCPUs with 7.5 GB of memory. For these experiments, we fix the message size at 32 bytes. The latency increases linearly with the message size, as we use more points to embed larger messages; i.e., a 32-byte message is one elliptic curve point, a 64-byte message is two elliptic curve points, etc.

Cryptographic primitives. Table 3 shows the performance of the cryptographic primitives used by Atom.

Group setup time. Table 4 shows the latency to generate an anytrust group of different sizes. The threshold key generation (DVSS [67]) is the dominating cost. For all group sizes of fewer than 64 servers, the setup takes less than two seconds. In practice, we expect this overhead to be even lower, since the servers can organize themselves into groups in the background as they mix the messages for the current round.

NIZKs vs. traps. To compare the performance of the two techniques for defending against malicious servers, we created a single anytrust group with 32 servers, and measured the time required to complete one mixing iteration. The number of messages varied from 128 to 16,384 messages, which is the expected range of message load per group. In the trap version of Atom, we accounted for the trap messages as well, which doubled the actual number of messages handled by each group. For example, if there are 1,024 groups and 2^{20}

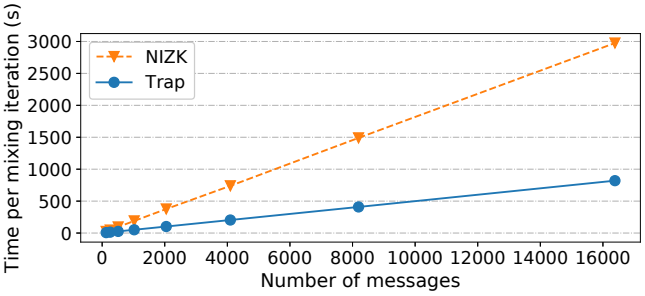

Figure 5. Time per mixing iteration for a single group of 32 servers as the number of messages varies.

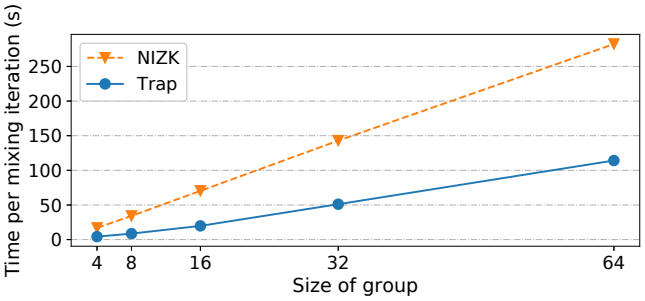

Figure 6. Time per mixing iteration for a single group when routing 1,024 messages as the group size varies.

messages, each group would handle 1,024 messages in the NIZK variant and 2,048 messages in the trap variant.

As shown in Figure 5, the mixing time of both modes increases linearly with the number of messages, since the mixing time largely depends on the number of ciphertexts each server has to shuffle and reencrypt. The NIZK variant takes about four times longer than the trap variant due to costly proof generation and verification. Based on this, we estimate that a full Atom network using NIZKs would be four times slower than a trap-based Atom network.

Group size. The size of each anytrust group in Atom depends on the security parameter and f, the fraction of servers that are adversarial. Figure 6 demonstrates the impact of group size on the mixing iteration time when the group handles 1,024 messages. For both schemes, the mixing time increases linearly with the group size, since each additional server adds another serial set of shuffling and reencryption operations. While our fault tolerance parameter h impacts the group size k as well, only $k - (h - 1)$ servers handle the messages in a given iteration since $k - (h - 1)$ servers is enough to decrypt the threshold encrypted ciphertexts.

Number of cores. The computations that the Atom servers perform are highly parallelizable, especially for the trap variant. Adding more cores to each server decreases the overall latency in proportion. To demonstrate this effect, we created an anytrust group of 32 servers using EC2 c4 nodes with 4, 8, 16, and 36 cores, and we routed 1,024 messages through them. Figure 7 shows the speed-up of different anytrust groups over the one consisting of only four core servers. The speed-up is nearly linear for the trap-variant, since majority of the load is parallelizable. The speed-up of the NIZK variant is sub-linear because the NIZK proof generation and verification technique we use is inherently sequential.

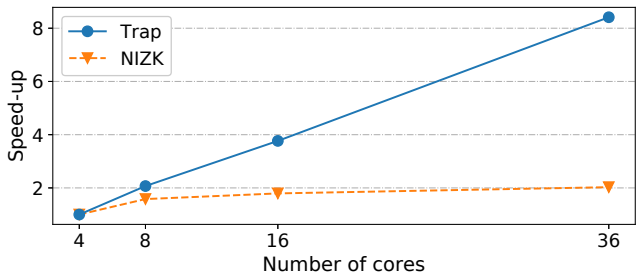

Figure 7. Speed-up of one mixing iteration of Atom as we increase the number of cores on a server. The baseline is when all servers have four cores.

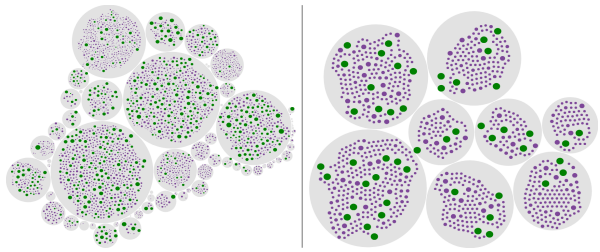

Figure 8. The Tor network topology [70] (left) and our network topology (right). The size of a node indicates its capacity; very high-capacity nodes are in green. In our topology, links within a cluster have 40 ms latency and across clusters have 80-160 ms latency.

6.2 Large-scale evaluation of Atom

Experimental setup. In this set of experiments, we used up to 1,024 various Amazon EC2 machines to test Atom's scalability and performance using the trap variant of Atom. In each experiment, 80% of the servers had 4 cores (c4.xlarge), 10% had 8 cores (c4.2xlarge), 5% had 16 cores (c4.4xlarge), and 5% had 32 cores (c4.8xlarge). We used the bandwidth statistics of the Tor network [3] as a proxy to chose the servers for our network[1]: 80% of the servers have less than 100 Mbps, 10% have between 100 Mbps and 200 Mbps, 5% have between 200 Mbps and 300 Mbps, and 5% have over 300 Mbps of available bandwidth. Figure 8 shows our network topology.

We picked the system parameters assuming $f = 20\%$ of the servers are malicious. We set up our groups to handle one server failure for each group (§4.5). Thus, we set the

[1]Statistics on core counts of Tor servers were not available.

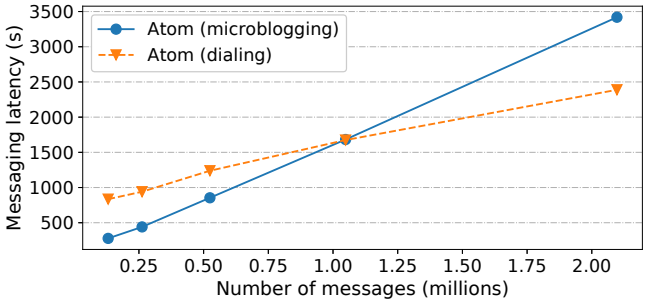

Figure 9. Latency of Atom for microblogging and dialing for varying number of messages. The latency increases linearly with the number of messages.

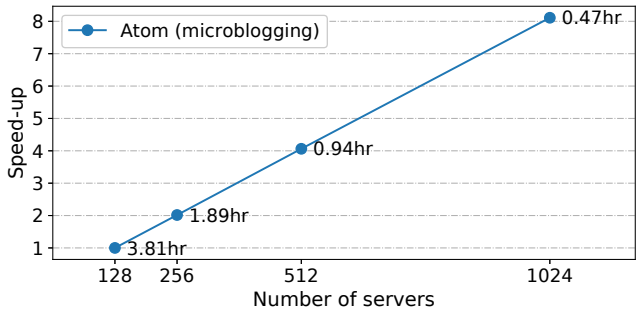

Figure 10. Speed-up of Atom networks of varying sizes relative to an Atom network of 128 servers. The speed-up is linear in the number of servers in the network.

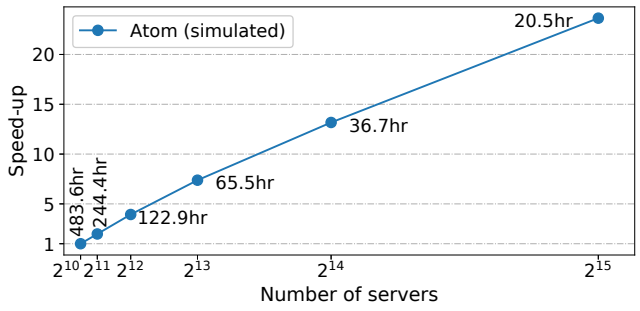

Figure 11. Simulated speed-up of Atom networks of varying sizes relative to an Atom network of 1,024 servers when routing a billion microblogging messages. At this scale, the speed-up is sub-linear in the number of servers.

group size to 33 servers, and required 32 out of the 33 servers to route the messages. Finally, we used $\mu = 13,000$, where μ is the average number of dummy messages by each server in an anytrust group for differential privacy [72]. Thus, on average, we expect about $32 \cdot \mu = 410,000$ dummy messages total in the network for anytrust groups of 32 servers.

We used $T = 10$ iterations with the square network for the evaluation. To measure the latency of one round of protocol run, we measure the time lapse between the moment that the first server in the first layer receives a message and the last server in the last layer outputs a message.

We answer the following questions in this section:

- Can Atom support a large number of users?
- How does Atom scale horizontally?
- How does Atom compare to prior systems?

Number of messages. We used 1,024 servers organized into 1,024 groups, and measured the latency as the total number of messages varied. As Figure 9 shows, the latency increases linearly with the total number of messages, since the number of messages handled by each group increases linearly with the total number of messages. The difference in the slope between the two applications is due to the smaller message size for dialing. For both applications, our prototype can handle over a million users with a latency of 28 minutes.

Horizontal scalability. To demonstrate that Atom scales horizontally, we measured the end-to-end latency for the network to route a million microblogging messages as the number of servers varied. As shown in Figure 10, the network speeds up linearly with the number of servers. That is, an Atom network with 1,024 servers is twice as fast as one with 512 servers.

We also simulated larger Atom networks to show further scalability. Here, we modified the implementation to model the expected latency given an input using values shown in Table 3, instead of actually performing the operations. We then had each physical server in our network emulate a large number of logical servers. Figure 11 shows the simulated latency when routing over a billion microblogging messages using larger Atom networks. Unlike the cases with less than 1,024 servers, we observed that the speed-up is slightly sub-linear in the number of servers. We believe there were two reasons for this. First, there are G^2 connections between two layers where G is the number of servers per layer. When there are tens of thousands of groups per layer, the number of connections became unmanageable for some servers. Second, the single trustee group experienced performance degradation. While the trustees could handle tens of thousands of TLS connections, the overhead of establishing TLS connection became non-negligible at this scale.

Comparison to prior work. Table 12 compares the performance of Atom to that of three prior works: Riposte [22], Vuvuzela [72], and Alpenhorn [50]. Riposte is an anonymous microblogging system that uses centralized anytrust servers. To compare Atom's microblogging capabilities, we used the fastest variant of Riposte which uses three 36-core machines configured to handle a million messages. As shown in Table 12, Riposte takes approximately 11 hours to anonymize a million messages. Atom can support a million microblogging messages in 3.8 hours with 128 servers and 28 minutes with

Table 12. Latency to support a million users. For microblogging, Atom is 23.7× faster than Riposte with 1,024 servers. Vuvuzela is 56× faster than Atom for dialing.

		Latency (min.)	
	Hardware Config.	*Microblog* (*Speedup vs. Riposte*)	*Dial* (*Slowdown vs. Vuvuzela*)
Atom	128×mixed	228.7 (2.9×)	225.1 (450×)
	256×mixed	113.4 (5.9×)	112.6 (225×)
	512×mixed	56.3 (11.9×)	55.5 (111×)
	1024×mixed	28.2 (23.7×)	27.9 (56×)
Alpenhorn [50]	3×c4.8xlarge	–	0.5 (1×)
Vuvuzela [72]	3×c4.8xlarge	–	0.5 (1×)
Riposte [22]	3×c4.8xlarge	669.2 (1×)	–

1,024 servers, which is 2.9× and 23.7× faster than Riposte. While Atom uses many more servers to achieve better performance, Riposte cannot take advantage of more servers without additional security assumptions. In particular, Riposte could scale by replacing each logical server with a cluster of physical servers. An adversary, however, still needs to only compromise one server from each cluster to break the security of the system.

Vuvuzela and Alpenhorn are private messaging systems with centralized anytrust servers that also support dialing. We show the dialing latency with one million users for the two systems when the network consists of three 36-core machines with 10 Gbps connection between them. These two systems are approximately 56× faster than Atom for mainly two reasons: (1) more efficient cryptography, as they use hybrid encryption, while Atom uses public key encryption, and (2) their machines are more powerful than the average machine in our network. Although Atom is slower, we still believe it is suitable for dialing. For example, Alpenhorn [50] suggests running a dialing round once every few hours due to client bandwidth limitations. Atom can support more than two million users on this time scale.

Atom also offers three concrete benefits over Vuvuzela and Alpenhorn. First, the bandwidth requirement for each server is significantly lower. Vuvuzela servers use 166 MB/sec of bandwidth for all servers [72], while Atom servers use less than 1 MB/sec of bandwidth. Second, Atom provides a clear way to scale further: adding more servers to our network will reduce the overall latency. Vuvuzela or Alpenhorn could replace each logical server with a cluster of servers to scale-out. However, similar to Riposte, the adversary only needs to compromise a server in each cluster to break the security of the system. Finally, Atom can provide additional privacy by preventing servers from tampering with honest users' messages. Vuvuzela and Alpenhorn do not aim to prevent servers from tampering with the users' messages, and as a result, a malicious server can drop all but one honest user's messages (the honest user will still be protected via differential privacy). In Atom, we guarantee that the users will enjoy anonymity among all honest users in addition to differential privacy.

7 Discussion and future work

We now discuss some aspects of Atom we did not consider in this paper and describe potential future work.

Estimated deployment costs. The deployment cost of a server depends on compute and bandwidth cost. When renting compute power from a commercial cloud service, the cost is usually fixed per hour of uptime. For example, if a volunteer wants to maintain a 100% up-time with a four core server on Amazon AWS, it would cost about $146/month, while a 36-core server would cost about $1,165/month as of September 2017.

The bandwidth cost depends on the number and the size of messages routed by the server, but we can estimate an upper bound by calculating the bandwidth requirement to rate-match the compute power of the server. Based on our microbenchmarks of the cryptographic primitives (Table 3), a four core server in the trap variant of Atom can reencrypt about 2,700 messages/second and shuffle about 9,200 messages/second when each message is 32 bytes. This translates to about 90KB/second and 300KB/second of bandwidth respectively. Assuming a constant flow of 300KB/second, the upper bound on the bandwidth cost would be about $7.20/month on AWS. For a 36-core server, this cost would scale linearly, and would cost about $65/month.

Denial-of-service. Our focus in this work was on providing scalable anonymity in the face of a near-omnipresent adversary. We explicitly left availability attacks out of scope. Our fault-tolerance mechanism (§4.5), however, can help defend against a small number of malicious failures as well: as long as there are fewer than h failures per group, benign or malicious, Atom can recover. Defending against a large scale DoS (e.g., in which the attacker takes more than half of the servers offline) remains an important challenge for future work. Proactively defending against malicious users in the trap variant is another important future work, as Atom can only retroactively identify malicious users after a disruption happens (§4.6).

Load balancing. In a real-world Atom deployment, there will be some variances in the capacity of the servers; e.g., number of cores, amount of available bandwidth, memory, etc. From a performance perspective, it would be beneficial to have the more powerful servers appear in more groups. Such non-uniform assignments of servers to groups, however, could result in an adversary controlling a full Atom group. We must therefore be careful when load-balancing servers, but the degradation in security may be worth the

performance gain in practice. Tor [29], the only widely deployed anonymity network, makes this trade-off: servers with more available bandwidth are more likely picked when forming a Tor circuit.

Intersection attacks by servers. Apart from the intersection attacks [28, 45] naturally occurring due to the changes in the set of Atom users, malicious servers in the trap variant of Atom could attempt to launch an intersection attack by selectively removing a user's ciphertexts. It is not possible, however, to remove both trap and message ciphertexts without getting caught, since the commitments of the traps are made available to all servers. Thus, the malicious server has to guess the real ciphertext to remove. As a result, the adversary performing such an attack on one user is caught with probability 50%, and this probability is amplified to $2^{-\kappa}$ for κ trials. Therefore, while Atom does not completely prevent intersection attacks by the servers, it does limit the number of times the adversary can attempt this attack.

8 Related work

Tor [29] is the only anonymous communication system in widespread use today. Like Atom, Tor scales horizontally. Unlike Atom, Tor aims to support low-latency real-time traffic streams, and Tor does not aim to defend against a global network adversary. Recent analysis of the Tor network suggests that even certain local adversaries may be able to deanonymize Tor users [17, 42, 60, 74].

Free-route mix-nets [27, 51, 57, 64] also scale horizontally. As with Tor, these systems do not provide strong anonymity properties in the face of powerful global adversaries. When the adversary can monitor the entire network and control some servers, the anonymity properties of these systems can degenerate to the set of users who share the same entry point to the network.

Mix-nets [18] and Dining Cryptographer networks (DC-Nets) [19] are the earliest examples of anonymity systems that provide protection against global adversaries. However, neither of these systems scales horizontally: mix-nets incur overhead linear in the number of servers, and DC-Nets incur overhead quadratic in the number of participants. Systems that build on these primitives [34, 48, 76] face similar problems when trying to scale. Riposte [22] is an anonymous microblogging system that uses techniques from private information retrieval [21] to scale to millions of users. Like a DC-Net, Riposte requires each server to perform work quadratic in the number of messages sent through the system.

The parallel mix-net of Golle and Juels [36] uses a distributed network of mix servers, similar to Atom. In this mix-net, Borisov showed that if the adversary controls some inputs and learns the output positions of the controlled inputs, then the adversary can infer some information about the messages sent by the honest users [15]. In contrast, knowing output positions of some users' messages in Atom does not result in anonymity loss for the other users.

Vuvuzela [72] and Alpenhorn [50] are two recent private-messaging systems that also provide *dialing* mechanisms that a user can use to establish a shared secret with another user. Both systems require all messages to pass through a centralized set of servers, making the systems scale only vertically. Moreover, contrary to Atom, malicious servers in these systems can drop all but one honest user's messages. In Atom, we ensure no honest users' messages are tampered with. Thus, Atom can provide anonymity in addition to differential privacy.

Vuvuzela also provides point-to-point metadata-hiding communication (not anonymity). Using Vuvuzela, two users who share a secret can communicate via the system without an adversary learning that these two users are communicating. Vuvuzela, however, cannot be used to anonymously organize protests, since the recipient of a message always knows its sender. Pung [8] addresses the same problem as Vuvuzela, but does so without the need for an anytrust assumption. Pung instead relies on computational private information retrieval, which escapes the need for trust assumptions but comes with significant computational costs [47].

Stadium [71] is a recent work that aims to horizontally scale Vuvuzela (i.e., private point-to-point communication) using distributed anytrust groups in a similar way to Atom. Stadium uses a system architecture inspired by the parallel mix-net [36] instead of permutation networks, and uses verifiable shuffle and cover traffic (dummy messages) to achieve a differential private notion of security. Stadium achieves lower latency than Atom by verifiable shuffling only the metadata of each message (e.g., a digest). The actual messages are encrypted using efficient hybrid encryption, and the servers after each iteration of mixing check that the metadata matches the ciphertexts. This strategy cannot be used in Atom since a user does not know the path that her message will take. Instead, Atom provides anonymous broadcasting without cover traffic and security based on indistinguishability of permutations at a higher latency.

Loopix [61] is a recent system that provides asynchronous bidirectional anonymous messaging, and scales horizontally. Loopix can provide low-latency communication using servers that insert small amount of delays before routing the messages. However, the security guarantees of Loopix degrade as the fraction of adversarial servers increases. In contrast, Atom provides a way to remain secure even when a large fraction of servers are actively malicious, at the cost of polylogarithmic increase in the latency due to larger group sizes. MCMix [7] is another system that provides bidirectional anonymous messaging, but using multiparty computation (MPC). MCMix achieves better performance than Atom

by defending against a weaker adversary. The MCMix prototype, for example, only supports three-server MPC that provides security against one passively malicious server.

The security analysis of Atom draws on the theoretical analysis of permutation networks. Permutation networks have long been studied as a way to permute a large number of elements using a network built of small components ("switches") [73]. In a 1993 paper, Rackoff and Simon [63] proposed building a distributed mix-net from a large permutation network. Their scheme required $O(\log^k n)$ iterations of mixing to mix n messages, where k was a "double-digit" number [24]. In Atom, we convert this theoretical result into a practical one. Abe also proposed building a distributed mix-net from a butterfly permutation network [4], though the construction had a flawed security analysis [5].

Recent theoretical work investigated the number of iterations of a butterfly network required so that a random setting of the switches produces a random permutation [52–54]. Czumaj and Vöcking [26] have recently argued that $O(\log^2(M))$ iterations are enough to generate an "almost" random permutation of M inputs. Czumaj [23] also studied randomly constructed networks and demonstrated that most networks of depth $O(\log^2(M))$ and width $O(M)$ produce good random permutations. Håstad studied a permutation network [40] based on shuffling the rows and columns of a square matrix. Any of these networks could be used as the underlying topology for an Atom network, but we focused on using the network by Håstad [40] due to its efficiency.

9 Conclusion

Atom is a traffic-analysis resistant anonymous messaging system that scales horizontally. To achieve strong anonymity and scalability, Atom divides servers into many anytrust groups, and organizes them into a carefully constructed topology. Using these groups, we design an efficient protocol for collectively shuffling and rerandomizing ciphertexts to protect users' privacy. We then propose two mechanisms, based on zero-knowledge proof techniques and trap messages, to protect against actively malicious servers. Finally, we provide a low-overhead fault-recovery mechanism for Atom. Our evaluation of Atom prototype on a distributed network consisting of over one thousand servers demonstrates that the system scales linearly as the number of participating servers increases. We also demonstrated that Atom can support more than a million users for microblogging and dialing. With its distributed and scalable design, Atom takes traffic-analysis-resistant anonymity one step closer to real-world practicality.

Acknowledgements

We thank Nirvan Tyagi, David Lazar, Riad Wahby, Ling Ren, and Dan Boneh for valuable feedback and discussion during this project. We also thank the anonymous reviewers, and our shepherd Peter Druschel. This work was supported in part by the NDSEG fellowship.

A Cryptographic details

We describe a modification to the ElGamal cryptosystem [32] for Atom. We work in a cyclic group $\mathbb{G}$ of order q with generator g in which the Decision Diffie-Hellman problem is hard [13]. The space of messages, ciphertexts, and public keys is $\mathbb{G} \cup \{\perp\}$, where $\perp$ represents a special null element.

- $(x, X) \leftarrow \mathsf{KeyGen}()$. Sample $x \leftarrow_R \mathbb{Z}_q$ and set $X = g^x \in \mathbb{G}$.
- $(R, c, Y) \leftarrow \mathsf{Enc}(X, m)$. Sample $r \leftarrow_R \mathbb{Z}_q$ and set $R \leftarrow g^r$. Set $c \leftarrow m \cdot X^r$, and $Y = \perp$. (R, c) forms the ElGamal ciphertext, and Y is a new element for Atom.
- $m \leftarrow \mathsf{Dec}(x, (R, c, Y))$. Return $m \leftarrow c/R^x$. The symbol "/" indicates multiplication by the inverse of the second operand. If $Y \neq \perp$, then the algorithm fails.
- $C' \leftarrow \mathsf{Shuffle}(X, C)$. To rerandomize a single ciphertext $(R, c, \perp)$ for public key X, sample $r' \leftarrow \mathbb{Z}_q$ and compute $(g^{r'} \cdot R, c \cdot X^{r'}, \perp)$. If $Y \neq \perp$, then the algorithm fails. To shuffle, rerandomize all ciphertexts and then permute them.
- $(R', c', Y') \leftarrow \mathsf{ReEnc}(x, X', (R, c, Y))$. If $Y = \perp$, then set $Y = R$ and $R = 1_G$, where 1_G is the multiplicative identity of G. The algorithm proceeds in two steps: First, remove a layer of encryption using x: $c_{\mathsf{tmp}} \leftarrow c/Y^x$. Then, reencrypt for X': sample $r' \leftarrow \mathbb{Z}_q$ and set $R' \leftarrow g^{r'} \cdot R$ and $Y' = Y$. Set $c' \leftarrow c_{\mathsf{tmp}} \cdot X'^{r'}$.
  Intuitively, Y holds the randomness used to encrypt for the current group, while R holds the randomness used to encrypt for the next group. By keeping both Y and R, the servers in the current group can decrypt out-of-order. In Atom, before the last server of a group forwards (R', c', Y') to the next group, it sets $Y' = \perp$; at this point, all layers of encryption by the current group have been peeled off, and c' is encrypted only under X', making Y' unnecessary.

We use a key encapsulation scheme with ElGamal for our IND-CCA2 encryption scheme for inner ciphertexts.

- $(x, X) \leftarrow \mathsf{KeyGen}()$. Sample $x \leftarrow_R \mathbb{Z}_q$ and set $X = g^x \in \mathbb{G}$.
- $(R, c) \leftarrow \mathsf{Enc}(X, m)$. Sample $r \leftarrow_R \mathbb{Z}_q$, and set $R \leftarrow g^r$. Generate a shared secret $k = X^r = g^{xr}$, and set $c \leftarrow \mathsf{AEnc}(k, m)$ where AEnc is an authenticated symmetric encryption scheme. For Atom, we used NaCl [12] for AEnc.
- $m \leftarrow \mathsf{Dec}(x, (R, c))$. Generate the shared secret $k = R^x = g^{xr}$. Set $m \leftarrow \mathsf{ADec}(k, c)$, where ADec is the decryption routine for AEnc.

Finally, we describe the NIZKs we use.

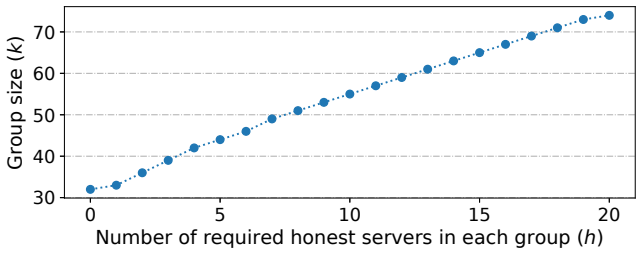

Figure 13. Required size of each group to maintain the security guarantees for different values of h for $f = 0.2$ and $G = 1,024$.

- $(c, \pi) \leftarrow \mathsf{EncProof}(pk, m)$. Compute the ciphertext $(g^r, m \cdot X^r, \perp) \leftarrow \mathsf{Enc}(X, m)$, and keep r. Pick a random $s \in \mathbb{Z}_q$. Then, compute $t = H(c \| g^s \| X)$, and $u = s + t \cdot r$, where H is a cryptographic hash function. Set $c = (g^r, m \cdot X^r, \perp)$ and $\pi = (g^s, u)$.
  To verify this proof, a verifier checks that $g^u = g^s \cdot g^{rt}$.
- $(c, \pi) \leftarrow \mathsf{ReEncProof}(sk, pk, m)$. We use the Chaum-Pedersen proof [20] without any modifications.
- $(C', \pi) \leftarrow \mathsf{ShufProof}(pk, C)$. We use the Neff verifiable shuffle [59] without any modifications.

For EncProof, the same proof π cannot be used for two different public keys, since the public key X is given as input to H when computing t. This prevents an adversary from copying the ciphertext and the NIZK submitted to one group, and submitting them to a different group.

B Many-trust group size

Atom handles server churns using many-trust groups, as described in §4.5. To tolerate up to $h - 1$ server failures in a group, each many-trust group needs at least h honest servers. We create such groups with high probability by increasing the group size. Let k be the group size, f be the fraction of malicious servers, and G be the number of groups in the network. Similar to the analysis done in §4.1 for the anytrust groups, we need

$$G \cdot \Pr[\text{fewer than } h \text{ honest servers in a group of } k \text{ servers}]$$
$$= G \cdot \sum_{i=0}^{h-1} \Pr[i \text{ honest servers in a group of } k \text{ servers}]$$
$$= G \cdot \sum_{i=0}^{h-1} \binom{k}{i} (1-f)^i f^{k-i} < 2^{-64}$$

Figure 13 shows the required size of the groups k as a function of h when $f = 0.2$, and $G = 1024$.

References

[1] Advanced crypto library for the Go language. https://github.com/DeDiS/crypto.

[2] Off-the-Record Messaging. https://otr.cypherpunks.ca/.

[3] Tor Metrics Portal. https://metrics.torproject.org.

[4] Masayuki Abe. 1999. Mix-networks on permutation networks. In *ASIACRYPT*. Springer, 258–273.

[5] Masayuki Abe and Fumitaka Hoshino. 2001. Remarks on mix-network based on permutation networks. In *PKC*. Springer, 317–324.

[6] Mehmet Adalier. 2015. Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256. (2015).

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

[8] Sebastian Angel and Srinath Setty. 2016. Unobservable Communication over Fully Untrusted Infrastructure. In *OSDI*. USENIX Association, GA, 551–569.

[9] Kevin Bauer, Damon McCoy, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. 2007. Low-Resource Routing Attacks Against Tor. In *WPES*.

[10] Stephanie Bayer and Jens Groth. 2012. Efficient Zero-knowledge Argument for Correctness of a Shuffle. In *EUROCRYPT*. Springer-Verlag, Berlin, Heidelberg, 263–280. https://doi.org/10.1007/978-3-642-29011-4_17

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

[12] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. 2012. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 159–176.

[13] Dan Boneh. 1998. The Decision Diffie-Hellman problem. In *International Algorithmic Number Theory Symposium*. Springer, 48–63.

[14] On Bitcoin as a public randomness source. https://eprint.iacr.org/2015/1015.pdf.

[15] Nikita Borisov. 2006. An Analysis of Parallel Mixing with Attacker-Controlled Inputs. In *International Workshop on Privacy Enhancing Technologies*. Springer Berlin Heidelberg, Berlin, Heidelberg, 12–25. https://doi.org/10.1007/11767831_2

[16] Justin Brickell and Vitaly Shmatikov. 2006. Efficient Anonymity-preserving Data Collection. In *KDD*. ACM, New York, NY, USA, 76–85. https://doi.org/10.1145/1150402.1150415

[17] Xiang Cai, Xincheng Zhang, Brijesh Joshi, and Rob Johnson. 2012. Touching from a Distance: Website Fingerprinting Attacks and Defenses. In *CCS*.

[18] David Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM* 24, 2 (Feb. 1981), 84–90. https://doi.org/10.1145/358549.358563

[19] David Chaum. 1988. The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. *Journal of Cryptology* 1, 1 (March 1988), 65–75.

[20] David Chaum and Torben P. Pedersen. 1993. Wallet Databases with Observers. In *CRYPTO*. 89–105.

[21] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private Information Retrieval. *J. ACM* 45, 6 (Nov. 1998), 965–981. https://doi.org/10.1145/293347.293350

[22] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An Anonymous Messaging System Handling Millions of Users. In *IEEE Symposium on Security and Privacy*. 321–338.

[23] Artur Czumaj. 2015. Random permutations using switching networks. In *STOC*. ACM, 703–712.

[24] Artur Czumaj, Przemyslawa Kanarek, Miroslaw Kutylowski, and Krzysztof Lorys. 1999. Delayed Path Coupling and Generating Random Permutations via Distributed Stochastic Processes. In *SODA*. 271–280.

[25] Artur Czumaj, Przemka Kanarek, Krzysztof Loryś, and Miroslaw Kutylowski. 2001. Switching networks for generating random permutations. In *Switching Networks: Recent Advances*. Springer, 25–61.

[26] Artur Czumaj and Berthold Vocking. 2014. Thorp Shuffling, Butterflies, and Non-Markovian Couplings. In *ICALP*. 344–355.

[27] George Danezis, Roger Dingledine, David Hopwood, and Nick Mathewson. 2003. Mixminion: Design of a Type III Anonymous Remailer Protocol. In *IEEE Symposium on Security and Privacy*. 2–15.

[28] George Danezis and Andrei Serjantov. 2004. Statistical disclosure or intersection attacks on anonymity systems. In *International Workshop on Information Hiding*. Springer, 293–308.

[29] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*. USENIX Association, 303–320.

[30] Morris J. Dworkin. 2014. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. (2014).

[31] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. 2012. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *IEEE Symposium on Security and Privacy*.

[32] Taher ElGamal. 1984. A public-key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*. Springer, 10–18.

[33] Jun Furukawa and Kazue Sako. 2001. An efficient scheme for proving a shuffle. In *CRYPTO*. Springer-Verlag, 368–387.

[34] Sharad Goel, Mark Robson, Milo Polte, and Emin Gün Sirer. 2003. *Herbivore: A Scalable and Efficient Protocol for Anonymous Communication*. Technical Report 2003-1890. Cornell University, Ithaca, NY.

[35] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game. In *STOC*. ACM, 218–229.

[36] Philippe Golle and Ari Juels. 2004. Parallel mixing. In *CCS*. ACM, 220–226.

[37] Philippe Golle, Sheng Zhong, Dan Boneh, Markus Jakobsson, and Ari Juels. 2002. Optimistic mixing for exit-polls. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 451–465.

[38] Edward Snowden: the whistleblower behind the NSA surveillance revelations. Accessed 2 November 2016, https://www.theguardian.com/world/2013/jun/09/edward-snowden-nsa-whistleblower-surveillance.

[39] Jens Groth and Steve Lu. 2007. Verifiable shuffle of large size ciphertexts. In *PKC*. 377–392.

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

[41] Jamie Hayes, Carmela Troncoso, and George Danezis. 2016. TASP: Towards anonymity sets that persist. In *Workshop on Privacy in the Electronic Society*. ACM, 177–180.

[42] Dominik Herrmann, Rolf Wendolsky, and Hannes Federrath. 2009. Website Fingerprinting: Attacking Popular Privacy Enhancing Technologies with the Multinomial Naive-bayes Classifier. In *ACM Workshop on Cloud Computing Security*. ACM, New York, NY, USA, 31–42. https://doi.org/10.1145/1655008.1655013

[43] Susan Hohenberger, Steven Myers, Rafael Pass, et al. 2014. ANONIZE: A large-scale anonymous survey system. In *IEEE Symposium on Security and Privacy*. 375–389.

[44] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. 2013. Users Get Routed: Traffic Correlation on Tor by Realistic Adversaries. In *CCS*.

[45] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. 2002. Limits of anonymity in open environments. In *International Workshop on Information Hiding*. Springer, 53–69.

[46] Shahram Khazaei, Tal Moran, and Douglas Wikström. 2012. A mix-net from any CCA2 secure cryptosystem. In *ASIACRYPT*. Springer, 607–625.

[47] Eyal Kushilevitz and Rafail Ostrovsky. 2000. One-way trapdoor permutations are sufficient for non-trivial single-server private information retrieval. In *EUROCRYPT*. Springer, 104–121.

[48] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. 2015. Riffle: An Efficient Communication System With Strong Anonymity. *PETS* 2016, 2, 115–134.

[49] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4/3 (July 1982), 382–401. https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/

[50] David Lazar and Nickolai Zeldovich. 2016. Alpenhorn: Bootstrapping Secure Communication without Leaking Metadata. In *OSDI*.

[51] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. 2013. Towards Efficient Traffic-analysis Resistant Anonymity Networks. In *SIGCOMM*. ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/2486001.2486002

[52] Ben Morris. 2008. The mixing time of the Thorp shuffle. *SIAM J. Comput.* 38, 2 (2008), 484–504.

[53] Ben Morris. 2009. Improved mixing time bounds for the Thorp shuffle and L-reversal chain. *The Annals of Probability* (2009), 453–477.

[54] Ben Morris. 2013. Improved mixing time bounds for the Thorp shuffle. *Combinatorics, Probability and Computing* 22, 01 (2013), 118–132.

[55] Mahnush Movahedi, Jared Saia, and Mahdi Zamani. 2014. Secure anonymous broadcast. *arXiv preprint arXiv:1405.5326* (2014).

[56] Steven J. Murdoch and Piotr Zieliński. 2007. Sampled Traffic Analysis by Internet-Exchange-Level Adversaries. In *PET*, Nikita Borisov and Philippe Golle (Eds.). Springer.

[57] Ulf MÂÍoller, Lance Cottrell, and Peter Palfrader. 2003. Mixmaster Protocol Version 2. (2003).

[58] Moni Naor and Moti Yung. 1990. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *STOC*. ACM, 427–437.

[59] C. Andrew Neff. 2001. A Verifiable Secret Shuffle and Its Application to e-Voting. In *CCS*. ACM, New York, NY, USA, 116–125. https://doi.org/10.1145/501983.502000

[60] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. 2011. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *WPES*. 103–114. https://doi.org/10.1145/2046556.2046570

[61] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. 2017. The Loopix Anonymity System. In *USENIX Security Symposium*. USENIX Association, Vancouver, BC, 1199–1216.

[62] Charles Rackoff and Daniel R Simon. 1991. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *CRYPTO*. Springer, 433–444.

[63] Charles Rackoff and Daniel R Simon. 1993. Cryptographic defense against traffic analysis. In *STOC*. ACM, 672–681.

[64] Michael K. Reiter and Aviel D. Rubin. 1999. Anonymous Web Transactions with Crowds. *Commun. ACM* 42, 2 (Feb. 1999), 32–48. https://doi.org/10.1145/293411.293778

[65] Dorothy Elizabeth Robling Denning. 1982. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[66] A Proposal for an ISO Standard for Public Key Encryption. Cryptology ePrint Archive, Report 2001/112. http://eprint.iacr.org/2001/112.

[67] Douglas R Stinson and Reto Strobl. 2001. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In *Australasian Conference on Information Security and Privacy*.

Springer, 417–434.

[68] E. Syta, P. Jovanovic, E. K. Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford. 2017. Scalable Bias-Resistant Distributed Randomness. In *IEEE Symposium on Security and Privacy*. 444–460. https://doi.org/10.1109/SP.2017.45

[69] NSA slide shows surveillance of undersea cables. https://www.washingtonpost.com/business/economy/the-nsa-slide-you-havent-seen/2013/07/10/32801426-e8e6-11e2-aa9f-c03a72e2d342_story.html. Accessed 26 October 2016.

[70] Tor Metrics. https://metrics.torproject.org/bubbles.html. Accessed 19 April 2017.. The visualization in Figure 8 is a modified version of the Tor Metric visualization. The original diagram carries a Creative Commons Attribution 3.0 United States License.

[71] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A Distributed Metadata-Private Messaging System. In *SOSP*. ACM.

[72] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *SOSP*. ACM, 137–152.

[73] Abraham Waksman. 1968. A permutation network. *J. ACM* 15, 1 (1968), 159–163.

[74] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. 2014. Effective Attacks and Provable Defenses for Website Fingerprinting. In *USENIX Security Symposium*. USENIX Association, San Diego, CA.

[75] Tao Wang and Ian Goldberg. 2013. Improved Website Fingerprinting on Tor. In *WPES*. ACM.

[76] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Dissent in Numbers: Making Strong Anonymity Scale. In *OSDI*. USENIX Association, Hollywood, CA, 179–182.

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

[78] Mahdi Zamani, Jared Saia, Mahnush Movahedi, and Joud Khoury. 2013. Towards Provably-Secure Scalable Anonymous Broadcast. In *FOCI*.