Scalable Shuffling with Butterfly Networks

Sam Anklesaria
Advised by Bryan Ford
April 30, 2015

Abstract

Numerous contexts in distributed systems such as group messaging and voting require the use of a multi-party shuffle, in which each node learns its position in a generated permutation without learning the positions of other nodes. Previously, common approaches to multi-party shuffles have scaled poorly; shuffle time scaled linearly with the number of nodes. We propose a shuffling technique that uses butterfly networks to integrate the results of several mix nets that scales polylogarithmically, and can run even faster if a small number of corrupted messages is acceptable.

1 Introduction

We live in an age in which anonymous communication online is difficult. Governments monitor Internet traffic on a massive scale. Private companies can make billions mining users’ browsing habits and linking their disparate Internet identities. Yet truly free speech requires anonymity, or those with unpopular ideas can be threatened with physical harm. Popular anonymous communication solutions such as Tor offer limited anonymity, but cannot protect users from sufficiently sophisticated adversaries. Additionally, Tor’s onion encryption mechanism is useful for point to point communication, but is not easily applied to situations where each client wants to broadcast messages to a large group, as is the case in group messaging or voting.

Alternatively, secure multi-party shuffles provide an approach to to anonymous communication which is resistant to traffic analysis and applies well to broadcasting. Multiple clients, each with their own input, want to agree on a random permutation of all their inputs without revealing which input came from which client. The shuffled inputs could be messages (in a group chat setting) or votes (in an electronic voting setting). Shuffles can also be used to bootstrap more efficient forms of anonymous communication—the DC-net based Dissent project requires an initial shuffle phase in order to assign slots of message bandwidth to clients anonymously [David Isaac Wolinsky, 2012].

If clients had access to a trusted third party, the problem would be trivial. All clients could submit their messages to the trusted node, which would shuffle the messages and distribute the results back to each client. Ideally, however, a multi-party shuffle would not require a central node, which could be compromised or go offline at any time. Instead, we would like to find an approach in which the clients submit their messages to a group of shuffler nodes, perhaps only a fraction of which are honest, which collectively produce a random permutation of the input elements—that is, a permutation which any node has probability roughly $1/n!$ of guessing—along with a proof that the output messages represent a permutation of the input messages.
2 Background

We use the general shuffling technique of a mix network. All clients onion-encrypt their messages and send them to some set starting nodes. These nodes remove a layer of encryption and forward the result to some intermediate nodes. This process continues until all layers of encryption have been removed, at which point all messages are forwarded to all clients. One of the simplest mix network approaches, similar to the strategy employed in [Brickell and Shmatikov, 2006] and the initial version of Dissent, is to pass all messages through a series of shuffle nodes. Clients wrap their messages in one layer of encryption per node. The first node receives all client messages and shuffles them randomly, removing a layer of encryption. It then forwards the shuffled messages to the next node in the order. Once all nodes have shuffled the messages and removed their component of the encryption, the last node in the ordering forwards the results to all clients.

The Brickell-style approach, however, does not scale well. Each node involved in the shuffle must permute all client messages. This becomes especially challenging if a verification mechanism is added to the protocol, requiring the construction of complex zero-knowledge proofs for each shuffle step. We propose a shuffling technique that distributes the computational burden of shuffling across many potentially untrusted nodes.

3 Cryptographic Preliminaries

Before describing the protocol, we will review some useful bits of cryptography.

3.1 ElGamal Pairs

ElGamal encryption provides an asymmetric key encryption algorithm for public key cryptography, and can be defined over any cyclic group $G$. If Alice wants to securely transmit a message to Bob, she can use Bob’s public, an element of $H$ of $G$ equal to a known base $g$ raised to a secret $h$ only Bob knows. She encodes her message as an element $m$ of $G$ and sends Bob the pair $(x, y)$, where $x = g^r$ for some newly random $r$ and $y = S^r m$. Bob can decrypt the message by finding $x^{-s}y$. We will assume that all client messages will be encoded as ElGamal pairs.

3.2 Butterfly Network

A butterfly network is an arrangement of switching nodes that can be used to permute $2^n$ elements. It consists of $l$ levels, with $n$ nodes on each level. Each switching node from level $i$ is connected to two different nodes on level $i + 1$. Two elements are given to each switching node at level 0. Each switching node randomly swaps its input elements, sending each input element to a different node in level 1. The process continues down the levels until the nodes at level $l - 1$ present their output: a permutation of the input elements. This shuffling scheme uses butterfly networks to merge the results of multiple shuffles together.

3.3 Neff Shuffles

As we have seen, mix nets rely on onion encryption to prevent nodes from linking a client’s message in the output permutation with its position in the input permutation (and exposing which client sent the message). We would like to verify, however, that each node in the mix net really performs a shuffle and decryption step to produce its output (instead of substituting its own fake messages). A zero-knowledge proof for this property is presented in [Neff, 2003], which allows observers to verify that one list of ElGamal pairs is an encrypted permutation of another. Specifically, suppose there are two sequences of pairs $(X_1, Y_1), \ldots, (X_k, Y_k)$ and $(\bar{X}_1, \bar{Y}_1), \ldots, (\bar{X}_k, \bar{Y}_k)$ representing input and output respectively. Neff shuffle proofs show there are some elements $\beta_1, \ldots, \beta_k$ and $\pi \in \sum_k$ such that, for known elements $g$ and $h$

$$\forall i \in [1, k], (\bar{X}_i, \bar{Y}_i) = (g^\beta_{\pi(i)} X_{\pi(i)}, h^{\beta_{\pi(i)}} Y_{\pi(i)})$$

2
3.4 Camenisch-Stadler Proofs
Camenisch-Stadler Proofs [Camenisch and Stadler, 1997] are proofs of discrete logarithms. Specifically, they prove statements of the form $P = a^x b^y c^z \ldots$, where the $P$ and the bases $a, b, \ldots$ are publicly known, but the exponents $x, y, \ldots$ are kept secret. They can also prove arbitrary conjunctions and disjunctions of such statements.

4 Shuffle Protocol
This protocol makes certain assumptions about the information available to each node. First, we assume that all $c$ clients and $n$ nodes participating in the shuffle are known in advance, along with the public keys of all nodes, and that all clients and nodes share a collectively generated random seed and parameters $l$ and $q$, the purpose of which will become clear shortly. This information could be established through a collective consensus protocol such as the one described in [Ewa Syta and Ford, 2015]. Second, we assume nodes address the liveness problem through some external means, such as in [Andreas Haeberlen and Druschel, 2007]. By itself, this protocol is easily stalled by malicious nodes which fail to send or respond to messages.

4.1 Shuffling with Butterfly Networks
We will start with a simplified form of the shuffle that assumes all nodes are trustworthy. Each node starts out with an equal share of the client message pairs, encrypted with the nodes’ public keys by the clients. Each node shuffles its input pairs and removes the layer of encryption added by the clients. For example, if the node in question had private key $s$, it would take pairs of the form $(x, y) = (g^{r_1}, g^{sr_1} m)$ and produce pairs of the form $(\bar{x}, \bar{y}) = (g^{r_1}, m)$ in a new permutation. Now, each node’s shuffled messages must be merged together. To this end, each node will divide its list of messages in half, directing each half to some other node. Each half will be re-encrypted with its target node’s public key before sending. These new nodes will each receive two half-size input lists, which they will shuffle together, decrypt, and re-encrypt the same way as before. After $l$ iterations of this divide and shuffle procedure, each node will send its shuffled messages back to the clients, which will concatenate together the output permutations of each node. This routing scheme can be thought of as a butterfly network of $l$ layers. Each node in level (in this case, iteration) $i$ has two edges to nodes in level $i + 1$. As an example, two layers of a simple butterfly network for nodes $A$, $B$, $C$ and $D$ are shown in the diagram below.

By [Czumaj and Vöcking, 2014], routing elements through a butterfly switching network with $\log^2(n)$ levels is sufficient to establish a fully random output permutation. Therefore, if the protocol is given a parameter $l \geq \log^2(n)$, the shuffle produces a random permutation of client messages. All nodes can agree on the topology of the butterfly network by using a predetermined pseudorandom number generator, seeded with the shared seed. Assigning edges between nodes in adjacent levels is equivalent to picking two different node permutations.

4.2 The Unverified Shuffle
Unfortunately, not all nodes may be trustworthy. Instead, we will assume that each node has some high probability $p$ of being trustworthy. If the $n$ nodes are randomly partitioned into $q$ quorums, this means the probability a quorum has no trustworthy nodes is $(1 - p)^{n/q}$. The number of nodes can be be adjusted to make this probability arbitrarily low. From now on, we will ignore this small probability, assuming that at
least one node in each quorum is trustworthy.

Now, the nodes in the butterfly network will consist of quorums rather than individual servers. Each of these quorums will have a designated order from starting node to ending node. The quorums complete a two phase shuffle process. In the first phase, the starting node for any given quorum \( t \) in level \( i \) will receive a list of pairs of the form \( (x, y) = (g^{r_1}, H^{r_1}m) \) from each of two quorums in the previous level \( i - 1 \), where \( H \) is the product of the public keys of the nodes in \( t \). If \( i = 0 \), these pairs are received from \( n/q \) of the clients. Each node from the starting node to the ending node will shuffle the pairs it receives, apply a new blinding factor, and forward them to the next node in the quorum order. Specifically, a pair \( (g^{r_1}, H^{r_1}m) \) would become \( (g^{r_1+\beta}, H^{r_1+\beta}m) \) for some random \( \beta \), and would be moved to a different index. When the last node in the order has shuffled the pairs, it forwards the result back to the first node, which begins the second phase.

In the second phase, each message is assigned a quorum in level \( i + 1 \) such that half the message are routed to some quorum \( u \) and the other half are routed to some quorum \( v \). Additionally, all message pairs are transformed into message triples, where the third element is the underlying cryptographic group’s identity element \( 1 \). Once again, each node from the starting node to the ending node will process the triples, forwarding the result to the next node in the order. This time, instead of shuffling the pairs, each node will remove its contribution to the encryption and add encryption for the message’s assigned quorum using a new blinding factor \( r_2 \). Specifically, if the node had private key \( s \), it would transform a triple \( (g^{r_1}, H^{r_1}m, k) \) assigned to a quorum with public key product \( A \) into the triple \( (g^{r_1}, A^{r_2}g^{-r_1s}H^{r_1}m, g^{r_2}) \). The ending node will forward the last two components of each triple to the message’s assigned quorum. The two stage process then repeats for the quorums in level \( i + 1 \).

### 4.3 The Verified Shuffle

The simplified protocol above is highly vulnerable to malicious nodes. Instead of shuffling the messages it receives in phase one, a malicious node might remove or duplicate some of its input messages. To guard against this possibility, we add a verification mechanism. In phase one, each node will use a Neff shuffle to permute its messages. In addition to the shuffled messages, it will forward the proof from the Neff shuffle, along with any proofs the node received itself, to the next node in the order. This next node must verify each of the proofs it received before shuffling. If any of the proofs fail, the node should ignore the input messages, as they may have been altered by a malicious or faulty node. After the ending node verifies all the proofs it receives, the proof from the quorum’s starting node will have been verified by every node in the quorum. As we assume at least one member of the quorum is honest, this is sufficient to confirm the validity of the first node’s shuffle, and we no longer need to transmit this proof. The other proofs have not been verified by every node in the quorum, however, and must be transmitted back to the starting node for verification in phase two. In the same fashion, each node in the second phase will verify all the shuffle proofs it receives, but will not forward the old proof it received to the next node in the sequence.

The same problem of vulnerability occurs with respect to decryption in second phase as well. In the simplified protocol above, a malicious node may remove or duplicate some of its input messages instead of partially decrypting and re-encrypting them without alerting other nodes to its misbehavior. To prevent this issue, a node sending to a quorum with collective public key \( A \) will generate a Camenisch-Stadler proof that \( \bar{y}/y = g^{-s}A^r \) for some blinding factor \( r \) and private key \( s \). In addition to the decrypted and re-encrypted messages remaining shuffle proofs, it will forward this proof, along with all the decryption proofs it received, to the next node in the sequence. As before, after the ending node verifies...
all the decryption proofs it receives, the decryption proof from the quorum’s starting node will have been verified by every node in the quorum. The other decryption proofs, however, will not have not been verified by every node in the quorum. They will need to be forwarded, along with half of the decrypted messages, to some quorum in the next level, which will continue the verification process in its phase one.

4.4 Randomized Routing

As described, this protocol requires a butterfly switching network of \( \log^2(n) \) levels to produce a fully random permutation. This cost can be reduced if we transform the random switching network into a random routing network. Instead of merely swapping messages in the left and right halves before transmitting each half of the messages to the next level, nodes can route the halves independently. Thanks to this extra bit of randomness, the modified protocol only requires \( \log(n) \) levels to produce a fully random permutation. Unfortunately, this scheme also means that inputs in separate halves could end up being sent to the same node in the next level. We handle this case by simply multiplying the conflicting group elements together. If the majority of the clients’ messages are the identity element (which can often be the case when only a small subset of available clients are communicating at any one time), collisions will not be problematic. Data loss would only occur if two non-identity messages collided.

5 Implementation

A prototype implementation was created in the Go programming language, consisting of roughly 2000 lines of Go code. This implementation has two front-ends: one that simulates communication on a single machine through the use of Go channels, and another that allows clients and servers on separate machines to communicate over the network. Both interfaces share a modular core that defines the shuffle protocol. This core exposes a generic interfaces for Shufflers (methods of permuting messages within a quorum) and Splitters (methods of distributing messages between quorums in the butterfly network). Currently, both Neff shuffles and simple binary shuffles are supported as Shufflers, and randomized switching and randomized routing are both supported as potential Splitters. The executables rely on an advanced cryptography library for the Go programming language developed in parallel with the Dissent project at Yale, which defines an abstract interface for performing operations on cryptographic groups. Preliminary tests of the shuffle implementation were conducted using the Emulab visualization system, although no comprehensive experiments have fully examined how the implementation scales.

6 Future Directions

This project began as an attempt to improve the performance of the setup phase of the Dissent anonymous message protocol. We hope some of the strategies presented here might find their way into the next Dissent prototype, and that this evaluation of scalable shuffle techniques can continue. One promising area we have so far ignored could have quorums calculate new permutations using fully homomorphic encryption instead of performing a sequence Neff shuffles. This would require less communication between nodes in each quorum, and could potentially be a faster approach. Fully homomorphic encryption has already been applied to shuffles with significant success in [Mahdi Zamani and Saia, 2015].

I also have not been able to benchmark the shuffle to the extent I would like. Although preliminary tests show verified shuffles of up to 200 clients taking under a minute, further work would conduct thorough experiments to determine how the runtime of a shuffle changes as the number of nodes participating increases.
References


