Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis

David Lazar, Yossi Gilad, and Nickolai Zeldovich, *MIT CSAIL*

https://www.usenix.org/conference/osdi18/presentation/lazar

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis

David Lazar, Yossi Gilad, and Nickolai Zeldovich
MIT CSAIL

Abstract

Karaoke is a system for low-latency metadata-private communication. Karaoke provides differential privacy guarantees, and scales better with the number of users than prior such systems (Vuvuzela and Stadium). Karaoke achieves high performance by addressing two challenges faced by prior systems. The first is that differential privacy requires continuously adding noise messages, which leads to high overheads. Karaoke avoids this using *optimistic indistinguishability*: in the common case, Karaoke reveals no information to the adversary, and Karaoke clients can detect precisely when information may be revealed (thus requiring less noise). The second challenge lies in generating sufficient noise in a distributed system where some nodes may be malicious. Prior work either required each server to generate enough noise on its own, or used expensive verifiable shuffles to prevent any message loss. Karaoke achieves high performance using *efficient noise verification*, generating noise across many servers and using Bloom filters to efficiently check if any noise messages have been discarded. These techniques allow our prototype of Karaoke to achieve a latency of 6.8 seconds for 2M users. Overall, Karaoke's latency is 5× to 10× better than Vuvuzela and Stadium.

1 Introduction

Text messaging systems are often vulnerable to traffic analysis, which reveals communication patterns like who is communicating with whom. Hiding this information can be important for some users, such as journalists and whistleblowers. However, building a messaging system just for whistleblowers is not a good idea, because using this system would be a clear indication of who is a whistleblower [9]. Thus, it is important to build metadata-private messaging systems that can support a large number of users with acceptable performance, so as to provide "cover" for sensitive use cases.

A significant limitation of prior work, such as Vuvuzela [26], Pung [1], and Stadium [25], is that they incur high latency. For example, with 2 million connected users, Vuvuzela has an end-to-end latency of 55 seconds, and the latencies of Pung and Stadium are even higher. Such high latencies hinder the adoption of these designs.

This paper presents Karaoke, a metadata-private messaging system that reduces latency by an order of magnitude compared to prior work. For instance, Karaoke achieves an end-to-end latency of 6.8 seconds for 2 million connected users on 100 servers (on Amazon EC2 with simulated 100 msec round-trip latency between servers), 80% of which are assumed to be honest, and achieves differential privacy guarantees comparable to Vuvuzela and Stadium. Furthermore, Karaoke can maintain low latency even as the number of users grows, by scaling horizontally (i.e., having independent organizations contribute more servers). Karaoke supports 16 million users with 28 seconds of latency, a 10× improvement over Stadium.

Achieving high performance requires Karaoke to address two challenges. The first challenge is that differential privacy typically requires adding noise to limit data leakage. Prior work achieves differential privacy for private messaging by enumerating what metadata an adversary could observe (e.g., the number of messages exchanged in a round of communication), and adding fake messages ("noise") that are mixed with real messages to obscure this information. This translates into a large number of noise messages that have to be added every round, and handling these noise messages incurs a high performance cost.

Karaoke addresses this challenge using *optimistic indistinguishability*. Karaoke's design avoids leaking information in the common case, when there are no active attacks. Karaoke further ensures that clients can precisely detect whether any information was leaked (e.g., due to an active attack), so that the clients can stop communicating to avoid leaking more data. This allows Karaoke to add fewer noise messages, because the noise messages need to mask fewer message exchanges (namely, just those where an active attack has occurred).

The second challenge lies in generating the noise in the presence of malicious servers. One approach is to require every server to generate all of the noise on its own, under the assumption that every other server is malicious [26]. This scheme leads to an overwhelming number of noise messages as the number of servers grows. Another approach is to distribute noise generation across many servers. However, a malicious server might drop the noise messages before they are mixed with messages from legitimate users. As a result, achieving privacy requires the use of expensive zero-knowledge proofs (e.g., verifiable shuffles) to ensure that an adversary cannot drop messages [25]. This approach reduces the number

of noise messages, but leads to significant CPU overheads due to cryptography.

Karaoke's insight is that verifiable shuffles are overkill: it is not necessary for all messages to be preserved, and it is not necessary to prove this fact to arbitrary servers. Instead, to achieve privacy, it suffices for each server to ensure that its noise is observed by all other servers. This can be done efficiently using Bloom filters, without having to reveal which messages are noise and which messages come from real users.

The contributions of this paper are as follows:

- The design of Karaoke, a metadata-private text messaging system that achieves an order of magnitude lower latency than prior work.
- Two techniques, optimistic indistinguishability and efficient noise verification, which allow Karaoke to achieve high performance.
- A privacy analysis of Karaoke's design that supports the use of these techniques.
- An experimental evaluation of a prototype of Karaoke.

One limitation of Karaoke is that it does not provide fault tolerance, since it requires all servers to be online. Handling server outages and denial-of-service attacks is an interesting direction for future work.

2 Related work

In this section, we compare Karaoke to prior work in two dimensions: privacy guarantees and the trade-off between scalability and server trust assumptions.

2.1 Privacy guarantees

Karaoke considers adversaries that control network links and some of the system's servers. This attacker model rules out systems based on Tor [7] such as Ricochet [3], due to traffic analysis attacks [5, 11, 18]. Loopix [20] is a recent system that delays messages and uses entropy [24] as a metric for reasoning about a user's anonymity set. However, Loopix does not provide any formal guarantees about privacy after users exchange multiple messages; it also requires users to trust a designated service provider [20: Table 1].

Some systems leak no information to the attacker, using techniques like DC-nets [28], Private Information Retrieval [1], or message broadcast [4]. Such systems provide the strongest form of privacy that users could hope for, but due to the quadratic overhead of these schemes in the number of users, their latency becomes high when supporting millions of users.

Karaoke achieves differential privacy for metadata-private messaging, much like Vuvuzela [26], Alpenhorn [15], and Stadium [25]. One key difference in Karaoke is that its design leaks no information about a user's traffic patterns in the common case, when there

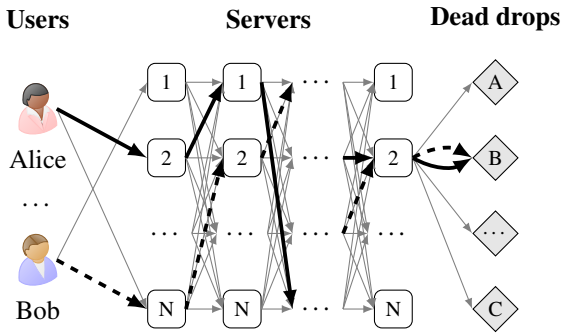

Figure 1: Overview of Karaoke's design.

are no lost messages, using the idea of optimistic indistinguishability. This allows Karaoke to add less noise for reaching the same privacy level as prior work [15, 25, 26], which improves performance.

Like Stadium, Karaoke is distributed over many machines, and must ensure that malicious servers do not compromise privacy. Stadium uses zero-knowledge proofs (e.g., verifiable shuffles) for this purpose, whereas Karaoke relies on more efficient Bloom filter checks.

2.2 Scalability vs. trust assumptions

Systems that assume the anytrust model (where all but one server may be malicious), such as Vuvuzela [26], Dissent [28], and Riposte [4], do not scale horizontally and cannot support the same magnitude of users as Karaoke.

One approach to horizontal scalability in metadata private messaging systems is to route messages through only a subset of all servers in the network, as in Loopix, Stadium, and Atom [14]. This requires trusting multiple servers to be honest, and introduces a tradeoff between the number of trusted servers (translating into the number of servers that process each message) and performance.

In Loopix every message is processed by a small number of servers (e.g., Loopix considers 3 or more servers to be a good choice [20: §4.3.1]). For privacy, Loopix requires that one of these servers is honest. However, if a significant fraction of servers are malicious, using a small number of servers means some users' messages will not be processed by any honest server. Karaoke ensures privacy with high probability by sending messages through more servers (e.g., 14 servers).

Atom [14] assumes that a fraction of the servers might be corrupt, and requires each message to be processed by many servers (hundreds). This leads to high latency, from 30 minutes to several hours. Karaoke also assumes that some fraction of servers are malicious. However it arranges its servers in a different, full-mesh topology, which allows it to achieve privacy while processing each message at fewer servers (e.g., 14 servers).

3 Overview

Figure 1 shows the main components of Karaoke. At the highest level, Karaoke consists of users, servers, and dead drops, similar to Vuvuzela and Stadium. All communication in Karaoke happens in rounds. In each round, users communicate by sending and receiving messages to and from dead drops. A dead drop is a designated location used to exchange messages. Dead drops are named by the server on which they are located, along with a pseudorandom identifier, and are not reused across rounds. When two users access the same dead drop, their messages are exchanged, and each user receives the other user's message. When two users want to communicate, they arrange to access the same dead drop (based on a shared secret). If a user is not communicating with anyone, he or she sends cover traffic to a randomly chosen dead drop.

The middle of the figure shows Karaoke's servers, labeled 1 through N, which are used to shuffle messages in order to hide information about which user is accessing which dead drop. The servers shuffle messages in *layers*, which are indicated by vertical groups in Figure 1, similar to a parallel mixnet [6, 8, 12, 21]. Each layer decrypts the messages (which are onion-encrypted) and re-orders them, so that the order of messages sent by a server does not correlate with the order in which the messages were received. Each server takes part in each layer; the figure depicts this by including each server in each layer. Between layers, servers exchange messages with one another.

The path of a message through the layers is chosen by the message sender at random. The message is onion-encrypted using the public keys of the servers on the chosen path, so that the message cannot be decrypted unless it passes through those servers. This ensures that an adversary cannot bypass the shuffling of the honest servers on the path of a message. Karaoke assumes that users know the public keys of all servers.

In Figure 1, Alice and Bob are communicating in a particular round. Their dead drop access paths are shown using bold arrows; solid for Alice and dashed for Bob. Alice and Bob send their messages to the same dead drop B on server 2. When the messages arrive at server 2, the server swaps them, and sends them back through the layers: Alice's message back to Bob along the reverse of the dashed arrows, and Bob's message back to Alice along the reverse of the solid arrows. This ensures server 2 does not know whose messages it swapped.

3.1 Goals and threat model

Karaoke's goal is to hide the communication patterns between users, so that an adversary cannot determine which users are communicating with one another. Karaoke does not hide information about which users are using Karaoke; an adversary can determine that a user is using Karaoke by observing a connection to one of Karaoke's servers. However, we hope that supporting a large number of users makes the mere act of using Karaoke less suspicious, similar to the argument by Dingledine et al. [7]. Karaoke also does not make availability guarantees; defending against DoS attacks is an interesting direction for future work.

In addition to Karaoke's privacy goals, Karaoke aims to achieve low latency for many users. This is important in order to enable broad adoption of Karaoke's design. Furthermore, Karaoke's goal is to provide horizontal scalability, so that Karaoke's operators can scale to more users over time by adding physical machines, thereby spreading the CPU and bandwidth requirements for operating Karaoke across more servers.

Karaoke assumes that an adversary has full control over the network and has compromised some number of servers and users' computers. Karaoke assumes that some fraction of servers (e.g., 80%) remains honest (not compromised), which we believe is achievable given leaked documents [19] and measurements of the Tor network [23, 27]. Karaoke hides communication patterns between users whose computers have not been compromised. If an adversary compromises a user's computer, the adversary can directly observe that user's activity, and Karaoke cannot provide any privacy guarantees. Karaoke makes standard cryptographic assumptions (the adversary cannot break cryptographic primitives), and assumes that Karaoke clients know the public keys of Karaoke servers.

We capture Karaoke's goal of hiding communication patterns using differential privacy [10], as in Vuvuzela and Stadium. Specifically, for a pair of users (call them Alice and Bob), Karaoke considers the probabilities of the observations that an adversary could make (e.g., observations of network traffic and observations from compromised servers), conditioned on Alice and Bob communicating or not communicating. Karaoke's differential privacy guarantee says that the probabilities of Alice and Bob communicating or not communicating, based on what the adversary observed, are close, and the ϵ and δ parameters control the degree of closeness (e^{ϵ} is a multiplicative factor and δ is an additive factor). The choice of the parameters is discussed in §6.1. Using differential privacy, Karaoke ensures that two users can always plausibly deny that they were communicating.

Since differential privacy is composable, a user can leverage this guarantee to reason about other plausible "cover stories." For example, if Alice was actually talking to Bob, she could instead claim she was talking to Charlie: the probability of her talking to Bob is within (ϵ, δ) of her not talking to anyone, which in turn is within (ϵ, δ) of her talking to Charlie, for a total of $(2\epsilon, 2\delta)$.

More formally, Karaoke treats the scenarios of two users communicating or not communicating with one another as "neighboring databases" in the context of dif-

ferential privacy. Since Karaoke relies on cryptography, Karaoke achieves *computational* differential privacy [17], rather than the perfect information-theoretic definition.

Karaoke's information leakage mostly comes from situations when a user's message is lost. This can occur either due to an active attack, or due to a long network outage (from which TCP cannot recover). Karaoke provides differential privacy for many rounds of message loss (hundreds, as discussed in §6.1). We expect users to avoid private conversations on highly unreliable networks; §7.6 provides some evaluation of network reliability.

Karaoke's design assumes that users can initiate conversations out-of-band. In other words, Karaoke hides metadata during a conversation. A complete messaging system would use Karaoke alongside a "dialing" protocol for one user to initiate a conversation with another user, and to establish a shared secret that is used to agree on a pseudorandom sequence of dead drops. The bootstrapping protocol would impose additional bandwidth and CPU costs for clients, but these costs are amortized over many conversation rounds. Alpenhorn [15] could serve as such a dialing protocol.

3.2 Privacy approach

Karaoke's design reveals two potential sources of information to the adversary: information about dead drop access patterns and information about how many messages were sent between servers across layers. In the rest of this section, we outline Karaoke's approach to hiding this information from the adversary.

Optimistic indistinguishability. To prevent the adversary from learning information based on dead drop access patterns, Karaoke's design strives to ensure that the dead drop access patterns look the same regardless of the communication pattern between users. Specifically, Karaoke requires that users always send *two* messages in a round. This allows a user to communicate with themselves if they are not otherwise communicating with a buddy, by arranging for their two messages to access the same dead drop. This gives the appearance of an active conversation to an adversary that is observing dead drop access patterns. If the user is communicating with a buddy, the user simply arranges for each of their messages to swap with a message from the buddy, using two different dead drops.

When the adversary is passive and there are no network outages, dead drop access patterns reveal no metadata about the communication of any pair of users. This is because, for a pair of users that might be either idle or chatting, there will be two dead drops, each of which is accessed twice. If messages are lost, an adversary may observe a dead drop with a single access, which may reveal some information. Karaoke addresses this through the use of noise messages, which we describe shortly. However, message loss is detectable in Karaoke because a user can simply look at the messages they receive back from the server to determine if any of their messages (or their buddy's messages) were lost.

Karaoke's "leakage-free" rounds allow it to improve performance by reducing noise and letting a client application decide how to handle leaky rounds. For example, the client application could choose to:

1. Alert the user, who could ignore it if their current conversation is not sensitive, or end the conversation if it is.

2. Retry the conversation after waiting (i.e., stopping the conversation but continuing to send cover traffic). This limits how quickly active attacks can learn information about the user.

3. Retry the conversation after switching to a new network (hopefully, one that is not under active attack).

These policies (or combinations of them) limit the rate at which an adversary can learn information through active attacks. This allows Karaoke to add less noise while still providing meaningful privacy guarantees.

Message swaps. A passive adversary in Karaoke can observe the number of messages sent between any two servers. To ensure that these observations do not reveal user metadata, Karaoke's topology is designed so that, for any pair of messages that traverse the same honest server in the same layer, an adversary cannot determine which path prefixes (i.e., paths leading up to this honest server) correspond to which path suffixes (i.e., paths taken by the messages after this honest server). In other words, the real scenario is indistinguishable from a scenario where the messages swap paths after the honest server.

The swapped paths correspond to the two neighboring databases. If Alice and Bob are communicating, then swapping the path suffix of one of Alice's messages with Bob's would mean that the two messages from Alice/Bob actually reach the same dead drop (so they are idle). Similarly, if Alice and Bob are idle, swapping path suffixes of two of their messages would mean that they are communicating.

This technique keeps the number of messages on each link identical regardless of whether message paths were swapped, thus preventing the adversary from learning useful information given the number of messages on every link.

Noise messages. Karaoke uses noise for two purposes: to protect dead drop access patterns in case messages are lost, and to enable message swaps. The noise takes the form of additional messages generated by the servers themselves. Each server generates messages to random dead drops, and routes those messages through random

paths in Karaoke's topology. These noise messages directly obscure the information available from the dead drop access patterns, because accesses by real users are now indistinguishable from accesses by noise messages.

Efficient noise verification. Some servers may be controlled by the adversary. It is crucial that these adversarial servers cannot subvert Karaoke's noise, either by generating insufficient noise in the first place, or by dropping noise messages as they traverse Karaoke's topology. Karaoke deals with the first problem by requiring all servers to generate enough noise to account for the possibility of malicious servers generating no noise at all.

To deal with the possibility of noise messages being dropped along the way, Karaoke uses *Bloom filters* [2] to efficiently check for the presence of noise at each layer. Each server at each layer in Karaoke's topology ensures that it has received all noise messages. It does so by computing a Bloom filter of all of the messages it has received, and sending this Bloom filter to all other servers. The other servers check whether the noise messages they generated appear in this Bloom filter. If any server indicates that their noise has been lost, the round is stopped.

Prior systems such as Stadium [25] deal with this problem by ensuring that no messages can be lost along the way. This requires expensive cryptographic techniques, such as verifiable shuffles. Karaoke's observation is that it suffices to ensure that noise messages are not lost. Using Bloom filters is a good choice because they do not require servers to reveal which messages were actually noise; the Bloom filter includes the set of all messages.

4 Design

This section describes Karaoke's design, starting with the overall structure and topology, and then describing the Karaoke client library and how Karaoke servers work.

4.1 Overall structure

Karaoke operates in rounds, which are driven by a coordinator. The coordinator is not trusted for privacy (its only job is to announce the start of a new round), but a malicious coordinator can impact the liveness of Karaoke. Round numbers must be strictly increasing, so the coordinator cannot trick clients into sending extra messages in a round, and if it announces a round multiple times, honest clients and servers will ignore it. Karaoke can distribute the user load over many coordinators (that are synchronized among themselves) since the coordinator's job is untrusted.

Karaoke's communication topology is shown in Figure 1. By using randomly chosen paths and exchanging messages at each layer, Karaoke provides a strong degree of mixing between all messages. Furthermore, Karaoke scales well with the number of servers, because each message is handled by a fixed number of servers (one per layer). As a result, adding more servers does not cause Karaoke to do more work overall.

```
def client_active(roundnum, myid, buddyid, buddysecret,
                  msg1, msg2):
  c1 = encrypt(buddysecret + "msg1" + myid, msg1)
  c2 = encrypt(buddysecret + "msg2" + myid, msg2)
  o1 = gen_onion(roundnum, myid, buddyid,
                 buddysecret + "onion1", c1)
  o2 = gen_onion(roundnum, myid, buddyid,
                 buddysecret + "onion2", c2)
  r1, r2 = karaoke_run_round(o1, o2)

  d1 = decrypt(buddysecret + "msg1" + buddyid, r1)
  d2 = decrypt(buddysecret + "msg2" + buddyid, r2)
  if d1 == None or d2 == None:
    raise("Message loss")
  return d1, d2

def client_idle(roundnum, myid):
  secret = random.secretvalue()
  c1 = random.ciphertext()
  c2 = random.ciphertext()
  o1 = gen_onion(roundnum, myid, myid + "dummy",
                 secret, c1)
  o2 = gen_onion(roundnum, myid + "dummy", myid,
                 secret, c2)
  r1, r2 = karaoke_run_round(o1, o2)
  if r1 != c2 or r2 != c1:
    raise("Message loss")

def gen_path(roundnum, rng):
  servers = get_servers_and_keys(roundnum)
  return [rng.choice(servers) for i in range(nlayers-1)]

# Choosing the last server to be one of the users' previous
# hops leads to more efficient noise generation.
def choose_last_srv(a, b):
  pair_choice = (a.id + b.id) % 2
  return sorted(a, b)[pair_choice]

def gen_onion(roundnum, myid, buddyid, secret, msg):
  mypath    = gen_path(roundnum, prng(secret + myid))
  buddypath = gen_path(roundnum, prng(secret + buddyid))
  drop_srv  = choose_last_srv(mypath[-1], buddypath[-1])
  drop_id   = prng(secret).rand128()

  onion = wrap((drop_id, msg), drop_srv)
  for srv in reversed(mypath):
    onion = wrap(onion, srv)
  return onion
```

Figure 2: Pseudocode for the Karaoke client.

4.2 Client

Figure 2 shows the pseudocode for the Karaoke client library. There are two modes of operation for the client: either the client is in an active conversation with a buddy, or the client is idle. In each round, the client must call either `client_active()` or `client_idle()`.

If the client is active, it must maintain a shared secret with the buddy, denoted `buddysecret` in the pseudocode. This secret should be established through a dialing protocol, such as Alpenhorn [15], and must evolve every round (e.g., by hashing it, or by using Alpenhorn's keywheel). Furthermore, if the client is active, it must pass two messages to `client_active()` that will be relayed to

the buddy; conversely, `client_active()` will return the buddy's two messages, if successful. Each message has a fixed size (256 bytes).

Onion generation. In each round, the client library generates two onions using `gen_onion()`. This function encapsulates a message `msg` in an onion encryption. The onion is sent towards a dead drop chosen pseudorandomly based on the shared `secret`, the ID of this user (`myid`), and the ID of the buddy (`buddyid`). For example, Figure 1's solid arrows indicate an onion sent by Alice to dead drop B on server 2. The payload, `msg`, is encrypted by the caller (specifically, by `client_active()`).

`gen_onion()` encrypts the message for each server in turn, using the public keys of the servers. The innermost encryption uses the key of the dead drop server, `drop_srv`. The other onion layers correspond to a path chosen by `gen_path()` using a pseudorandom number generator.

One subtle detail is that the dead drop server, `drop_srv`, is chosen deterministically in `gen_onion()` to be one of the servers from the two users' paths in the previous layer (either `mypath[-1]` or `buddypath[-1]`). This is an optimization that reduces the degrees of freedom in Karaoke, and thus allows Karaoke to generate noise efficiently, as we will discuss in §4.3.

The dead drop ID, `drop_id`, is chosen pseudorandomly based on the shared `secret`. This ensures that an adversary cannot learn any information by observing the accessed dead drop IDs (since the secret changes every round), yet the two users agree on the same dead drops.

Active conversation. When a client is in an active conversation, `client_active()` exchanges two messages with the user's buddy. It does so by first encrypting the two messages, `msg1` and `msg2`, to produce two ciphertexts `c1` and `c2`. The pseudocode uses + to derive subkeys from the `buddysecret` master key. `client_active()` then calls `gen_onion()` twice, with two subkeys derived from `buddysecret` (appending the strings `onion1` and `onion2` respectively). These onions are then passed to `karaoke_run_round()`, which sends the onions through Karaoke's server topology and waits for responses, if any.

Once `client_active()` receives the responses, it must verify that no message loss took place—that is, that the adversary did not block either of this user's two messages, or the buddy's two messages. `client_active()` checks for this by ensuring that it receives two ciphertexts that properly decrypt (using authenticated encryption). If an adversary dropped one of the messages from this client, `karaoke_run_round` will return `None`, causing the decryption check to fail. If an adversary dropped one of the messages from the buddy, the last server hosting the dead drop will observe just one message reaching the dead drop and echo back this client's message in response, which will similarly cause the decryption check to fail (because the message is not encrypted using the subkey generated with `buddyid`). If no message loss took place, `client_active()` returns the decrypted messages.

Sending a message back to the user in case of message loss is important since if there is a conversation between Alice and Bob, and an adversary drops Bob's message, then one naive outcome might be that now Alice receives nothing in response in that round. This would be quite unfortunate: the adversary will know Bob was talking to Alice! By echoing back the message, the last server sends at least some (fixed-size) data towards Alice, so that an adversary cannot tell that Alice was Bob's conversation partner. (To be precise, a random response would also suffice in this case.) Intermediate servers similarly enforce that every request must receive a response, in case the last server was malicious.

Idle client. When there is no active conversation, Karaoke's client library ensures that the externally observable behavior, from the adversary's perspective, remains identical. `client_idle()` does so by generating random ciphertexts, `c1` and `c2`, which should be indistinguishable from ciphertexts that would have been generated by `client_active()`. `client_idle()` chooses a random `secret`, and constructs two onions, `o1` and `o2`, simulating a conversation between users `myid` and `myid+"dummy"`.

Much like `client_active()`, `client_idle()` needs to check for message loss. It does so by ensuring that it receives `c2` and `c1` respectively in response to its onions.

Handling message loss. In Karaoke, message loss can leak information to an adversary, and thus reduce the degree of privacy that the user can expect. Karaoke detects such events, which allows the client application built on top of the library from Figure 2 to avoid excessive privacy loss. Specifically, Karaoke's client closes any active conversation after encountering message loss. This prevents an adversary from dropping a user's messages in many rounds to learn additional information. Other policies for dealing with message loss can be implemented that balance usability and privacy, as outlined in §3.

Karaoke should rarely lose messages, because IP packet loss in the network is handled by TCP (see §7.6). Thus, the primary source of false positives are long-lived network outages. We recommend that users stop sensitive conversations when their network becomes unreliable (regardless of whether it is the result of an attack).

4.3 Server

Figure 3 presents the pseudocode for Karaoke's server. The pseudocode focuses on the processing of onions from clients to the dead drops, as well as the generation and verification of noise messages. Not shown is the logic for setting up per-round public keys (signed with a long-term private key of each server), accepting inputs from users in

```
def process_layer(roundnum, layer, inputs):
  msgs = [decrypt(srvkey[roundnum], msg)
          for msg in inputs]
  msgs = dedup(msgs)
  if layer == 0:
    msgs += generate_noise(roundnum)
  else:
    bloom = bloomfilter.new(inputs)
    for srv in get_servers_and_keys(roundnum):
      if srv.rpc("check_bloom", roundnum,
                 layer, bloom) != True:
        raise("Lost noise, halting round")

  outgoing = collections.defaultdict(list)
  for m in msgs:
    outgoing[m.next_hop].append(m)

  for srv, q in outgoing:
    srv.rpc("enqueue_batch_for_process_layer",
            roundnum, layer+1, shuffle(q))

def check_bloom(roundnum, layer, bloom):
  caller = get_rpc_caller()
  for m in noise msgs routed via caller at layer:
    if m not in bloom:
      return False
  return True
```

Figure 3: Pseudocode for Karaoke's server.

the first layer, exchanging the messages that are addressed to the same dead drop in the last layer, and sending the responses back to the clients.

Layer processing. Each server uses the `process_layer()` function shown in Figure 3 to process the set of input messages at a given layer. In the first layer, the server collects input messages from clients until the round coordinator kicks off the round processing. In subsequent layers, each server waits to receive inputs from every server in the previous layer.

Layer processing starts by decrypting the inputs and de-duplicating them. It is important to remove duplicates (and to ensure the ciphertexts are not malleable), because otherwise an adversary could tag a victim's message by replicating it several times and looking for which message appears to be replicated at the end of Karaoke's topology.

Noise. The next step of layer processing involves ensuring that the necessary noise is present. In the first layer, each server generates noise; subsequent layers use Bloom filter checking to ensure that noise has not been dropped by malicious servers.

Noise generation. At the start of every round, each server generates noise. The goal of noise messages is to mask dead drop access patterns in the case of message loss, meaning that legitimate user messages did not form a pair of accesses to the same dead drop. In this case, an adversary observes some number of dead drops with two accesses, and some number with just a single access (due to a non-paired message). This translates into the two kinds of noise messages generated by Karaoke: "singles" (noise message that generates a single dead drop access), and "doubles" (a pair of noise messages that generates a double access to the same dead drop).

Karaoke's threat model assumes that some servers may be malicious, but it is not known *a priori* which servers are malicious. An adversary could use a malicious server to trace back the source of a dead drop access to the last honest server in the path. Thus, as we show in our analysis [16], it is important that all outgoing links from every server carry an adequate number of noise messages, since every link could potentially be the outgoing link from the last honest server on some message's path.

Like Stadium [25], Karaoke uses the Poisson distribution to sample noise messages. This distribution is a good fit for distributed noise generation for two reasons. First, it allows precisely sampling a non-negative integer for the number of messages, even if the distribution mean is low. Second, the sum of many small Poisson samples is also a Poisson distribution, simplifying the analysis.

Let N be the number of servers, and l be the length of Karaoke paths (`nlayers` in the pseudocode). Our topology provides N^l possible routes, which makes it computationally cumbersome to sample for every route individually, and inefficient, since there are only $(l-1)\cdot N^2$ communication links in the entire system (there are $l-1$ transitions between layers, and in each transition each server is connected to all others). We would ideally like to just sample the amount of noise on every link.

To generate the singles noise, a server begins by sampling the noise for the links to the last layer of servers (layer l), and samples how many messages go over each of the N^2 links in that phase. For each link, the server samples from the Poisson distribution, with mean λ_1. The server then sums them up to find how many of its noise messages need to leave each server in the previous layer. The server then samples again, to decide how many noise messages travel on each link to the servers in the previous layer $(l-1)$. Of course, there will likely be a mismatch; i.e., a server in layer $l-1$ has to distribute a different number of messages than it receives. In this case, the server just adds incoming or outgoing noise messages to match the other by adding extra noise messages and distributing them uniformly among all links. Karaoke continues in this fashion until it reaches the first layer. The number of these extra messages is unlikely to be large, because it is simply the difference between two samples from the same Poisson distribution. Overall, each server samples $(l-1)\cdot N^2$ times from the noise distribution to assign single-access noise.

To generate doubles noise, the server performs a similar procedure to the one described above. Notice that in the last layer we only iterate over the $N^2/2$ possible pairs of links that output messages to the same dead-drop hosting

server ($N^2/2$ is the number of possible second-to-last-hop pairs of servers, since order does not matter). This is because the dead-drop hosting server is chosen deterministically by `gen_onion()` based on `choose_last_srv()`. Similarly to the above, for each such pair, we sample noise from the Poisson distribution with mean λ_2. The result denotes the number of pairs of messages, where one message is routed on each link. In all layers before the last one, the procedure for generating double-access noise is exactly the same as the single-access noise case described above.

Preserving noise. In layers after the first one, the servers must ensure that noise messages have not been dropped by a malicious server from a previous layer. Karaoke servers do this by computing a Bloom filter [2] over all of the messages received by that server in a particular layer. Each server then sends its Bloom filter to all other servers to check whether their noise appears to be present. As long as all servers indicate that their noise is present, this server can assume that no noise messages from honest servers have been dropped, and proceed with processing the layer.

The only queries that matter are an honest relay checking with an honest noise-sender. A malicious noise-sender does not matter since it can send zero noise. A malicious relay does not matter since it can relay messages even if noise is missing. We incorporate both of these in determining how much noise is needed (generating extra noise to account for malicious servers that generate zero noise).

At each hop, one encryption layer of the message is decrypted. If an adversary does not know a server's private key, the adversary cannot predict the decryption result (it looks pseudorandom, since the onion contains another encrypted message). A malicious server that refuses to forward a message cannot guess the decrypted version of that message after the next honest hop. Thus, the adversary cannot fill in another message that will "look like" the dropped message in the Bloom filters of subsequent honest servers. Karaoke's topology and parameters ensure at least two honest servers in every path (with high probability); see analysis in §5.

To check whether noise messages are present, a server runs `check_bloom()`. This function must first determine which noise messages were routed through the calling server at a given `layer`, and second, determine the ciphertext representation of the onion that would be seen by that server at that layer. Finally, `check_bloom()` verifies that all of those ciphertexts are in the Bloom filter, without disclosing which messages are noise and which are real.

The Bloom filter has false positives, which may lead `check_bloom()` to falsely conclude that a noise message is present. In Karaoke, it is up to the server running `process_layer()` to construct the Bloom filter with adequate parameters to achieve suitably false positive rate. If the server running `process_layer()` is malicious, it can construct a Bloom filter with 100% false positive rate. However, such a malicious server could also ignore the result of `check_bloom()` altogether.

The probability of not detecting n discarded noise messages shrinks exponentially with n, since messages are independently pseudorandom (see above). This allows Karaoke to use relatively small Bloom filters (with 10% false positive rate) and yet ensure that no more than a few noise messages may be lost (for $n = 20$ the probability of missing detection is 10^{-20}). Karaoke generates a few extra noise messages to account for the possibility that several might be lost without detection (but not more).

Noise verification involves an all-to-all communication, but does not lead to quadratic bandwidth requirements as the number of servers grows. This is because increasing the number of servers would proportionally reduce the size of the Bloom filters, since the Bloom filters represent only those messages that are handled by a particular server. Other horizontally scalable systems have similar phases. For example, Stadium [25], which most closely related to Karaoke, includes an all to all distribution between "input chains" to "output chains"; in Stadium, this phase involves cryptographic computations (signature verification and NIZKs). Although in Karaoke the all-to-all communication happens at every hop, the number of hops is fixed so the overhead of Karaoke is expected to remain much smaller than Stadium even for large deployments.

5 Analysis

This sections shows that Karaoke achieves its privacy goal (§3.1), which is captured by the following theorem.

Theorem 1. Karaoke is ϵ, δ-differentially private with respect to the following neighboring databases: (1) Alice is talking with another user Bob, and (2) Alice is idle.

Proof sketch. We show Theorem 1 holds in the analysis below by the following argument. We begin by showing that Karaoke servers maintain noise messages in the system (§5.1). Next, we analyze optimistic indistinguishability, showing that in the common case Karaoke leaks no communication metadata under passive attacks (§5.2). Optimistic indistinguishability has one caveat: the attacker may launch active attacks to learn some information about the communication patterns of some users. We use differential privacy to reason about the amount of information leaked to the attacker under this scenario (§5.3).

The differential privacy parameters (ϵ and δ), the singles and doubles noise (λ_1 and λ_2), and the number of rounds k for which this theorem holds are discussed in §6.1. An extended technical report [16] provides detailed proofs.

5.1 Efficient noise verification

For Karaoke's privacy guarantees to hold, it is crucial to prevent the attacker from discarding noise messages generated by the honest servers. Karaoke identifies when noise messages are discarded using Bloom filter checks (§4.3). Bloom filters, however, allow for false positives, so a few noise messages might be dropped even if the Bloom filter check shows they are present. With a false positive rate p, the probability that k lost noise messages go undetected is p^k. Even with a relatively high $p = 10\%$, it is sufficient to increase the mean of the single- and double-access noise distributions (λ_1 and λ_2, from §4.3) by just $\frac{20}{h}$ (where h is the number of honest servers) to ensure Karaoke keeps adequate noise with probability $> 1 - 10^{-20}$.

Adjusting the Bloom filter size allows Karaoke to control the false positive rate, but the size of the Bloom filter reveals the number of messages processed by a server. This is acceptable, as the rest of Karaoke's analysis does not rely on the total number of messages being hidden.

5.2 Optimistic indistinguishability

We continue our analysis by showing that combining noise with Karaoke's routing topology prevents metadata leakage. That is, if the two messages from Alice and the two messages from Bob route through the system, then it is very likely to be completely indistinguishable whether they exchange messages with each other (active mode) or with themselves (idle mode). We begin our analysis by explaining the conditions under which optimistic indistinguishability holds, and then evaluate the probability for these conditions to hold considering a passive adversary.

5.2.1 Avoiding metadata leakage

Karaoke's optimistic indistinguishability stems from the following theorem:

Theorem 2. Assume that two messages a and b, from honest senders (users or servers), route through an honest server s^i at layer i. Denote the two message routes by $\langle s_a^1, \ldots, s^i, \ldots, s_a^l \rangle$ and $\langle s_b^1, \ldots, s^i, \ldots, s_b^l \rangle$. Then it is equally likely, given the attacker's observations of the inter-server links and malicious intermediary servers (i.e., observations on all but the last server), that a routes through $\langle s_a^{i+1} \ldots, s_a^l \rangle$ and b routes through $\langle s_b^{i+1} \ldots, s_b^l \rangle$ or vice-versa.

Proof. Since s_i is honest, its shuffle permutation is unknown to the adversary. Each message in Karaoke takes an independent route. Denote the outgoing links from server s^i that a and b take by l_1, l_2, and the attacker's observations on outgoing links from s^i by O. It holds that $\Pr[a \text{ takes } l_1 \mid O] = \Pr[b \text{ takes } l_1 \mid O]$ and that $\Pr[a \text{ takes } l_2 \mid O] = \Pr[b \text{ takes } l_2 \mid O]$. Therefore,

$$\Pr[a \text{ takes } l_1 \wedge b \text{ takes } l_2 \mid O] = \Pr[a \text{ takes } l_2 \wedge b \text{ takes } l_1 \mid O]$$

Furthermore, since messages are onion-encrypted, the bit-level representations of messages a and b forwarded by s^i are indistinguishable from random. As a result, an adversary cannot distinguish whether a travels over the link $s^i \to s_a^{i+1}$ and b over $s^i \to s_b^{i+1}$ or vice-versa.

Assume that a and b swap the suffix of their routes following layer s^i. Since the two messages swap routes, the number of messages on each following link remains the same (and the messages themselves are indistinguishable from one another because they are onion-encrypted). Therefore all of the attacker's observations on inter-server links remain the same, regardless of whether the two messages were swapped. □

Theorem 2 allows us to swap between two messages. However, it requires that the two swapped messages route through the same honest server. The next theorem, which follows from Theorem 2, extends this observation and shows that even messages with non-intersecting routes can be indistinguishably swapped, with the help of noise messages.

Theorem 3. Let a and b be two messages that route through $\langle s_a^1, \ldots, s_a^l \rangle$ and $\langle s_b^1, \ldots, s_b^l \rangle$ respectively. Let n_0 and n_1 be two other messages from honest participants that route through $\langle s_{n_0}^1, \ldots, s_{n_0}^l \rangle$ and $\langle s_{n_1}^1, \ldots, s_{n_1}^l \rangle$. Assume that there exists some i_0 and j_1 such that $s_{n_0}^{i_0} = s_a^{i_0}$ and $s_{n_0}^{j_1} = s_b^{j_1}$, where the servers $s_a^{i_0}$ and $s_b^{j_1}$ are honest and $i_0 < j_1$. This means that, for some layer i_0, n_0 and a route through the same honest server, and for some layer j_1, n_0 and b route through the same honest server. Similarly, assume there exists some i_1 and j_0 such that $s_{n_1}^{i_1} = s_b^{i_1}$ and $s_{n_1}^{j_0} = s_a^{j_0}$, where the servers $s_b^{i_1}$ and $s_a^{j_0}$ are honest, $i_1 < j_0$, $i_0 < j_0$, and $i_1 < j_1$. Under these conditions, and using observations from network links and intermediary servers, it is indistinguishable whether the messages took their actual routes or the following alternative routes:

a routes via $\langle s_a^1, \ldots, s_a^{i_0}, s_{n_0}^{i_0+1}, \ldots, s_{n_0}^{j_1-1}, s_b^{j_1}, \ldots, s_b^l \rangle$
b routes via $\langle s_b^1, \ldots, s_b^{i_1}, s_{n_1}^{i_1+1}, \ldots, s_{n_1}^{j_0-1}, s_a^{j_0}, \ldots, s_a^l \rangle$
n_0 routes via $\langle s_{n_0}^1, \ldots, s_a^{i_0}, s_a^{i_0+1}, \ldots, s_a^{j_0-1}, s_{n_1}^{j_0}, \ldots, s_{n_1}^l \rangle$
n_1 routes via $\langle s_{n_1}^1, \ldots, s_b^{i_1}, s_b^{i_1+1}, \ldots, s_b^{j_1-1}, s_{n_0}^{j_1}, \ldots, s_{n_0}^l \rangle$

Proof. Applying Theorem 2 four times on the following arguments gives the result:

1. on messages a, n_0 at honest server $s_a^{i_0}$
2. on messages b, n_1 at honest server $s_b^{i_1}$
3. on messages a, n_1 at honest server $s_a^{j_0}$
4. on messages b, n_0 at honest server $s_b^{j_1}$

Figure 4 illustrates these four swaps (where message $a = a_1$ and message $b = b_0$). □

Given four messages a, b and n_0, n_1 the attacker cannot identify, using observations on network links and malicious intermediary servers, whether the messages take one route where a, b end up on servers s_a^l, s_b^l and n_0, n_1 end up on servers $s_{n_0}^l, s_{n_1}^l$ or they take an alternative route where a, b reach s_b^l, s_a^l and n_0, n_1 reach $s_{n_1}^l, s_{n_0}^l$. However, if the last servers (s_*^l) turn out to be malicious, then the attacker might still distinguish between the two scenarios. To see why, consider the case where n_0 is a double-access noise message and its pair routes through an all-malicious route. In this case, the attacker can observe the difference between the two alternative scenarios because the last server on n_0's route would have actually received n_1 instead of n_0 and therefore would observe one less double access and two more single accesses if n_0 and n_1 were to swap (i.e., using the alternative routes in Theorem 3). The next theorem describes how messages between two honest users can be swapped without leaking information to the attacker, when n_0 and n_1 are single-access noise messages.

Theorem 4. If the premise for Theorem 3 holds for two user-messages a and b and two single-access noise messages n_0 and n_1, then it is indistinguishable whether a routes through $\langle s_a^1, \ldots, s_a^l \rangle$ and b through $\langle s_b^1, \ldots, s_b^l \rangle$, or a routes through $\langle s_a^1, \ldots, s_a^{i_0}, s_{n_0}^{i_0+1}, \ldots, s_{n_0}^{j_1-1}, s_b^{j_1}, \ldots, s_b^l \rangle$ and b routes through $\langle s_b^1, \ldots, s_b^{i_1}, s_{n_1}^{i_1+1}, \ldots, s_{n_1}^{j_0-1}, s_a^{j_0}, \ldots, s_a^l \rangle$.

Proof. Applying Theorem 3 shows that given just observations from network links and intermediary servers, an adversary cannot determine which message takes what route. We now focus on the last servers of each message route. Assume that they are all malicious and allow the attacker to observe the dead-drop access patterns. The last server on n_0's route, in the alternative routing scheme, would have received n_1 (after all four swaps); see illustration in Figure 4. Since n_1 and n_2 are two single access noise messages, generated by honest servers, the malicious last server would observe in both cases an encrypted message (that was encrypted by an honest server) reaching a dead drop by itself. Similarly this holds for the last server on n_1's route. The user messages a and b would both reach encrypted to a double-access dead drop (since the attacker is passive, the paired message reaches the dead drop too). So both cases are indistinguishable. □

We refer to two messages a and b for which there exists two single-access noise messages n_0 and n_1 that satisfy the premise of Theorem 4 as *indistinguishably swappable*. We next use Theorem 4 to analyze Karaoke's privacy guarantees.

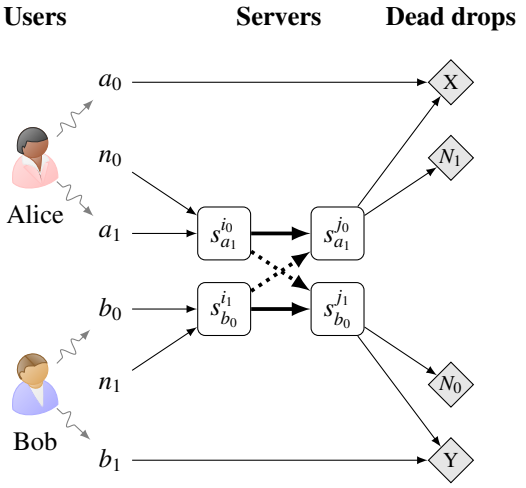

Figure 4: An illustration of Karaoke's optimistic indistinguishability: an adversary cannot determine whether Alice and Bob are communicating via dead drops X and Y. Straight lines represent links (potentially across multiple intermediate servers) that an adversary can track. Servers $s_{a_1}^{i_0}$, $s_{b_0}^{i_1}$, $s_{a_1}^{j_0}$, and $s_{b_0}^{j_1}$ are honest. Solid bold lines indicate the actual path taken by messages a_1 and b_0. Dotted bold lines indicate the actual path taken by messages n_0 and n_1. An adversary cannot distinguish whether a_1 and b_0 took the solid or dotted bold lines. Squiggly lines indicate users generating two messages in a round.

5.2.2 Alice talking with Bob, and claims "idle"

Consider two users, Alice and Bob, who may be talking with each other or idle. Alice sends two messages a_0, a_1 and Bob sends b_0, b_1. If Alice and Bob communicate, then Alice's a_0 meets Bob's b_0 at the dead drop, and a_1 meets b_1 at a different (and independently chosen) dead drop. If they do not communicate, then a_0 meets a_1 at a dead drop and so do b_0 and b_1.

Theorem 5. If one of the pairs of messages $\langle a_0, b_1 \rangle$ or $\langle a_1, b_0 \rangle$ is indistinguishably swappable, then it is indistinguishable whether Alice is talking to Bob or they are both idle.

Proof. To understand why this theorem holds, consider Figure 4. Assume without loss of generality that the premise holds for the pair of messages $\langle a_1, b_0 \rangle$. Applying Theorem 4 on $\langle a_1, b_0 \rangle$, it is therefore indistinguishable whether a_1 routes to dead drop X and b_0 routes to dead drop Y or vise versa. In the first scenario a_0 meets b_0 at dead drop Y and a_1 meets b_1 at dead drop X, so Alice and Bob are talking. In the second (indistinguishable) scenario it is actually a_0 that meets a_1 at dead drop X and b_0 that meets b_1 at dead drop Y so Alice and Bob are idle. Importantly, it does not matter what route Alice and Bob's other messages, a_0 and b_1, take; the servers handling these messages may all be malicious. □

Our technical report [16] analyzes the probability with which optimistic indistinguishability holds. For example, with $N = 100$ servers, out of which $h = 80$ are assumed

honest, a chain length of $l = 14$, and where each honest server generates single-access noise with mean $\lambda_1 \geq 0.5$ (so the mean of single-access noise on each link is $h\lambda_1 = 40$), the probability that optimistic indistinguishability holds is at least $1 - 5 \cdot 10^{-14}$.

5.2.3 Alice idle, and claims "talking with Bob"

Theorem 6. If the premise for Theorem 4 holds for at least one of the message pairs $\langle a_0, b_0 \rangle$, $\langle a_0, b_1 \rangle$, $\langle a_1, b_0 \rangle$, $\langle a_1, b_1 \rangle$, then it is indistinguishable whether Alice is talking to Bob or they are both idle.

When Alice and Bob are idle, a_0, a_1 and b_0, b_1 travel to the same dead drop. It is therefore sufficient to indistinguishably swap one of four options: a_0 with b_0, or a_0 with b_1, or a_1 with b_0, or a_1 with b_1 (rather than two options as in §5.2.2: a_0 with b_1, or a_1 with b_0). This gives an even higher probability of achieving indistinguishability.

5.3 Message loss and differential privacy

An active attacker can discard user messages before Karaoke unlinks them from their senders (e.g., before the first layer, as users submit messages to Karaoke). This might prevent Karaoke from "indistinguishably swapping" messages as required for our analysis in the passive case (§5.2). We now analyze this scenario. The technical report [16] includes the proofs for the theorems below.

Consider a user Alice and an active attacker who tries to learn whether she is talking with Bob.

Theorem 7. The active attacker's best strategy (leaking the most information) is to either discard both messages from Alice, or both messages from Bob.

Intuitively, the theorem holds since if the attacker discards both messages from Alice or both messages from Bob, there are no messages to swap with so optimistic indistinguishability never holds. The following theorem holds when the attacker is active:

Theorem 8. Karaoke is ϵ, δ-differentially private in the face of message loss (e.g., due to active attackers), if both user messages route through at least two honest servers.

The conditional in Theorem 8 holds with overwhelming probability in the route length parameter l. For example, with a route length $l = 14$, assuming 80% of the servers are honest, this conditional holds with probability $1 - 2 \cdot 10^{-8}$ (which is folded into the differential privacy δ parameter of Karaoke).

6 Implementation

Karaoke is implemented in 4000 lines of Go code, compiled with Go 1.11. Onion decryption dominates the CPU costs of our prototype and is implemented in native `amd64` assembly, provided by Go's NaCl library. The servers use the gRPC library over TLS for communication. We use streaming RPCs and batching RPCs together to reduce latency. Karaoke issues RPCs over multiple TCP connections to improve throughput.

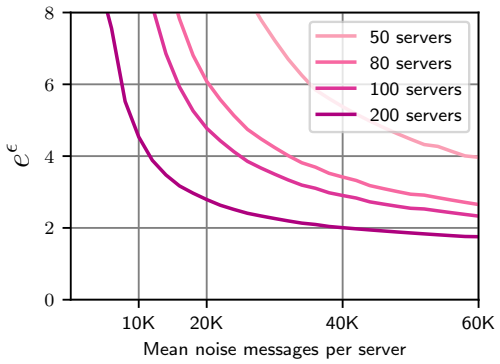

Figure 5: e^ϵ as a function of the number of noise messages per server per round, for $\delta = 10^{-4}$, $h = \lfloor 0.8N \rfloor$, and $l = 14$.

6.1 Parameter selection

We would like Karaoke to provide good privacy guarantees even after users communicate via Karaoke for a long time. We target $\epsilon = \ln 4$ and $\delta = 10^{-4}$ after 10^8 rounds of communication, of which 245 rounds encounter message loss during a sensitive conversation.

Figure 5 plots the expected number of noise messages that an honest server generates in a round, and the resulting e^ϵ privacy guarantee (with a fixed $\delta = 10^{-4}$ after 10^8 communication rounds with 245 rounds of message loss), for deployments of $N = 50, \ldots, 200$ servers where we assume $h = \lfloor 0.8N \rfloor$ servers are honest, and route length $l = 14$. For example, in our configuration using 100 servers, each server generates an average of $N^2\lambda_1 + N^2\lambda_2 = 25\text{K}$ noise messages per round. Computing the data in Figure 5 required the use of composition over multiple rounds [10, 13].

As we evaluate in §7.6, 245 rounds of message loss is about an order of magnitude higher than the number of expected losses due to network outages in a year. Karaoke could achieve the same privacy guarantee under more active attacks by adding more noise.

7 Evaluation

We quantitatively answer the following questions:

- Can Karaoke achieve low latency for many users?
- Can Karaoke scale to more users by adding servers while maintaining the same low latency?
- How is Karaoke's performance affected by the fraction of honest servers?
- How important are Karaoke's techniques for achieving low latency?

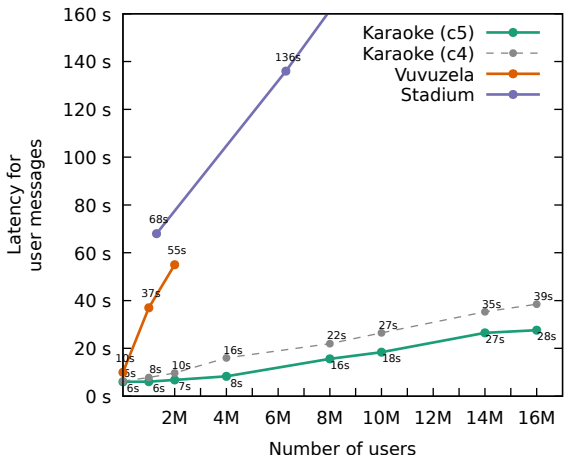

Figure 6: End-to-end latency of user messages with a varying number of users. Vuvuzela is running with 3 servers; Karaoke and Stadium are both running with 100 servers.

- How often would network problems cause Karaoke users to observe message loss?

7.1 Experimental setup

To answer the above questions we ran our prototype on Amazon EC2 using `c5.9xlarge` instances (36× Intel Xeon 3.0 GHz cores with 72 GB of memory and 10 Gbps links). We ran experiments using VMs in the same data center to save on AWS bandwidth costs. Realistically, Karaoke would be deployed on servers in different countries (or trust zones). For example, we envision some fraction of the servers running in the US and the rest running in different countries in Europe. We simulate this topology by adding 100ms of round-trip network latency (the round-trip time from the east coast of the US to Europe) to each VM using the `tc qdisc` command.

We simulate millions of users by having servers generate extra messages in the first layer (to avoid the cost of launching many more client VMs). The extra messages are pre-generated (before the round starts) so that server CPU costs are not muddled by what would normally be client CPU costs.

An additional VM is used to run a coordinator server. This server has two jobs: it starts rounds across all Karaoke servers and injects probe messages into each round to measure the end-to-end latency of the round.

Unless specified otherwise, our experiments assume that 80% of the servers are honest, which translates into a topology with 14 layers. Karaoke's Bloom filters are tuned for a 10% false positive rate, as discussed in §5.1.

7.2 Karaoke achieves low latency

To evaluate Karaoke's end-to-end latency we ran an experiment using 100 Karaoke servers. Figure 6 shows the results. For comparison, we also include the latency of Vuvuzela and Stadium as reported in their papers which provide privacy comparable to Karaoke. The Vuvuzela and Stadium results used `c4.8xlarge` VMs, so we also measured Karaoke's performance on this less powerful instance type. Stadium's performance was achieved using 100 servers with a chain length of 9. Vuvuzela used only 3 servers because its performance does not increase with the number of servers.

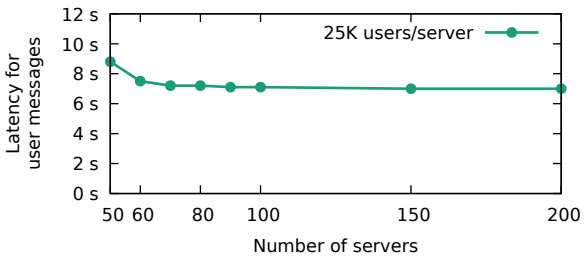

Figure 7: End-to-end latency of user messages with 25K users per server, with a varying number of servers.

The results show that with 2M users Karaoke achieves 5× lower latency than Vuvuzela, and 8× lower latency than Stadium (using the weaker `c4` instances). Furthermore, the slope of the Karaoke line in Figure 6 shows that Karaoke scales better with more users than either Vuvuzela or Stadium. Karaoke's scaling is better than Vuvuzela because only a fraction of Karaoke servers are involved in handling the messages from every additional user, whereas every Vuvuzela server must handle every additional user's messages. Karaoke's scaling is better than Stadium because Stadium must perform expensive zero-knowledge proofs for every additional user message, whereas Karaoke's marginal cost are just in onion decryption and network bandwidth. For instance, Karaoke achieves 10× lower latency than Stadium with 16M users.

7.3 Scaling by adding servers

The previous subsection shows that Karaoke's latency increases as more users join the system. This is unavoidable if the number of servers is fixed. Ideally, Karaoke would be able to support additional users without increasing latency by adding a proportional number of servers. To evaluate if this is the case, we measured the end-to-end latency of Karaoke with a varying number of servers and a proportional number of users (25K users per server).

Figure 7 shows the results, which indicate that Karaoke can maintain low latency for an increasing number of users by adding more servers to the system. Karaoke's latency goes down slightly as the number of servers grows because it requires less noise, as shown in Figure 5.

7.4 Fraction of honest servers

Figure 8 shows the number of layers required to achieve Karaoke's privacy guarantees with a varying fraction of honest servers, and the impact that increasing the number of layers has on end-to-end latency. The results show Karaoke's tradeoff between lower latency and fewer trusted servers. When fewer servers are assumed honest,

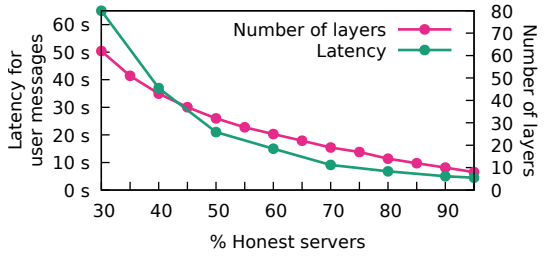

Figure 8: End-to-end latency for 2M user messages and 100 servers with a varying fraction of honest servers. The right y-axis shows the required number of layers to achieve privacy for a given fraction of honest servers.

each honest server has to create more noise to compensate for the possibility of malicious servers not sending any noise. Karaoke achieves acceptable latency for text messaging even if only 60% of the servers are honest. On the other hand, Karaoke would not be a good fit if only 30% of the server were honest.

7.5 Importance of techniques

To demonstrate the importance of Karaoke's key techniques (optimistic indistinguishability and using Bloom filters for efficient noise verification), we consider the performance of Karaoke without these techniques. In the absence of optimistic indistinguishability, Karaoke would need to add ~320K noise messages per server per round to achieve the same level of privacy. This translates into an increase in latency from 6.8s to 31s for 2 million users.

In the absence of Bloom filters, Karaoke could use verifiable shuffles similar to Stadium. For 6 million users and 100 servers, each Stadium server spends 6s generating verifiable shuffles and another 2s verifying shuffles at each hop in the network. Karaoke, on the other hand, spends 250ms generating and checking Bloom filters at each hop. Using verifiable shuffles in Karaoke would increase Karaoke's overall latency by about 2 minutes (8 seconds for each of Karaoke's 14 hops). This shows that both techniques are crucial for Karaoke's performance.

7.6 Leakage due to network issues

Karaoke's design avoids leaking information when the network is well-behaved, by arranging for all dead drop access to occur in pairs. However, network issues could result in some information being leaked if some dead drop accesses are no longer paired. Karaoke runs over TCP so momentary packet loss will not prevent message delivery. On the other hand, if clients can not communicate with the Karaoke servers for an extended period of time, they will be unable to submit their message into a round.

To estimate how often this might happen, we performed an experiment by probing a Karaoke server every 2 minutes for a day from 100 machines using RIPE ATLAS [22], which provided machines distributed across the globe that communicate with our server. Each probe consisted of 3 ping packets, spaced 1 second apart. The experiment generated 71,194 probe results, of which 70,106 received responses to all 3 pings, 991 received 2 responses, 60 received 1 response, and 37 received no responses (indicating a complete loss of network connectivity). The complete losses of network connectivity occurred in "bursts," where a machine experienced complete loss of connectivity for several adjacent two-minute intervals. The complete losses were encountered by 8 machines (7 of them observing one "burst" and one observing two "bursts").

These results suggest that a Karaoke client could encounter approximately 9 message loss events over 100 days, or about 33 such events per year. (Since Karaoke clients switch to idle mode after detecting message loss, only the first loss in a burst matters for this analysis.) This compares favorably with the message loss that Karaoke's parameters can handle (245, as discussed in §6.1).

8 Conclusion

Karaoke improves the latency of metadata-private text messaging by almost an order of magnitude compared to prior work. Karaoke also scales well with the number of users and the number of servers, maintaining its low latency. To achieve its performance, Karaoke introduces a new design, exchanging messages between each server in multiple layers, as well as two key techniques. *Optimistic indistinguishability* allows Karaoke to achieve perfect privacy with high probability in case no messages from the user (and their peer) are lost, and allows clients to detect message loss. *Efficient noise verification* allows Karaoke to generate noise messages across many servers, and to use efficient Bloom filter checks to prevent adversaries from discarding the noise. We hope that Karaoke's low latency will bring metadata-private messaging closer to widespread adoption.

Acknowledgments

Thanks to Derek Leung, Georgios Vlachos, Adam Suhl, and the PDOS group for their helpful comments and suggestions. Thanks also to the anonymous reviewers and our shepherd, Ranjita Bhagwan. This work was supported by NSF awards CNS-1413920 and CNS-1414119, and by Google.

References

[1] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 551–569, Savannah, GA, Nov. 2016.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] J. Brooks et al. Ricochet: Anonymous instant messaging for real privacy, 2016. https://ricochet.im.

[4] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, pages 321–338, San Jose, CA, May 2015.

[5] G. Danezis. Statistical disclosure attacks: Traffic confirmation in open environments. In *Proceedings of the 18th International Conference on Information Security*, pages 421–426, Athens, Greece, May 2003.

[6] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *Proceedings of the 24th IEEE Symposium on Security and Privacy*, pages 2–15, Oakland, CA, May 2003.

[7] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, Aug. 2004.

[8] R. Dingledine, V. Shmatikov, and P. F. Syverson. Synchronous batching: From cascades to free routes. In *Proceedings of the Workshop on Privacy Enhancing Technologies*, pages 186–206, Toronto, Canada, May 2004.

[9] Z. Dorfman. Botched CIA communications system helped blow cover of Chinese agents. *Foreign Policy*, Aug. 2018. https://foreignpolicy.com/2018/08/15/botched-cia-communications-system-helped-blow-cover-chinese-agents-intelligence/.

[10] C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[11] Y. Gilad and A. Herzberg. Spying in the dark: TCP and Tor traffic analysis. In *Proceedings of the 12th Privacy Enhancing Technologies Symposium*, pages 100–119, Vigo, Spain, July 2012.

[12] P. Golle and A. Juels. Parallel mixing. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, pages 220–226, Washington, DC, Oct. 2004.

[13] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. In *Proceedings of the 32nd International Conference on Machine Learning*, Lille, France, 2015.

[14] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 406–422, Shanghai, China, Oct. 2017.

[15] D. Lazar and N. Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 571–586, Savannah, GA, Nov. 2016.

[16] D. Lazar, Y. Gilad, and N. Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis (extended technical report). Technical report, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, Oct. 2018. Also available at https://vuvuzela.io/karaoke-extended.pdf.

[17] I. Mironov, O. Pandey, O. Reingold, and S. Vadhan. Computational differential privacy. In *Proceedings of the 29th Annual International Cryptology Conference (CRYPTO)*, pages 126–142, Santa Barbara, CA, Aug. 2009.

[18] S. J. Murdoch and G. Danezis. Low-cost traffic analysis of Tor. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*, pages 183–195, Oakland, CA, May 2005.

[19] National Security Agency. Tor stinks. The Guardian, Oct. 2013. https://www.theguardian.com/world/interactive/2013/oct/04/tor-stinks-nsa-presentation-document.

[20] A. M. Piotrowska, J. Hayes, T. Elahi, S. Meiser, and G. Danezis. The Loopix anonymity system. In *Proceedings of the 26th USENIX Security Symposium*, pages 1199–1216, Vancouver, Canada, Aug. 2017.

[21] C. Rackoff and D. R. Simon. Cryptographic defense against traffic analysis. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing (STOC)*, pages 672–681, 1993.

[22] RIPE Network Coordination Centre. RIPE Atlas, May 2018. https://atlas.ripe.net/.

[23] A. Sanatinia and G. Noubir. Honey onions: A framework for characterizing and identifying misbehaving Tor HSDirs. In *Proceedings of the 2016 IEEE Conference on Communications and Network Security (CNS)*, pages 127–135, Oct. 2016.

[24] A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In *Proceedings of the Workshop on Privacy Enhancing Technologies*, pages 41–53, San Francisco, CA, Apr. 2002.

[25] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 423–440, Shanghai, China, Oct. 2017.

[26] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, Monterey, CA, Oct. 2015.

[27] P. Winter, R. Köwer, M. Mulazzani, M. Huber, S. Schrittwieser, S. Lindskog, and E. Weippl. Spoiled onions: Exposing malicious Tor exit relays. In *Proceedings of the 14th Privacy Enhancing Technologies Symposium*, pages 304–331, Amsterdam, Netherlands, July 2014.

[28] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson. Dissent in numbers: Making strong anonymity scale. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Hollywood, CA, Oct. 2012.