Yodel: Strong Metadata Security for Voice Calls

David Lazar, Yossi Gilad[†] and Nickolai Zeldovich
MIT CSAIL and Hebrew University of Jerusalem[†]

Abstract

Yodel is the first system for voice calls that hides metadata (e.g., who is communicating with whom) from a powerful adversary that controls the network and compromises servers. Voice calls require sub-second message latency, but low latency has been difficult to achieve in prior work where processing each message requires an expensive public key operation at each hop in the network. Yodel avoids this expense with the idea of *self-healing circuits*, reusable paths through a mix network that use only fast symmetric cryptography. Once created, these circuits are resilient to passive and active attacks from global adversaries. Creating and connecting to these circuits without leaking metadata is another challenge that Yodel addresses with the idea of *guarded circuit exchange*, where each user creates a backup circuit in case an attacker tampers with their traffic. We evaluate Yodel across the internet and it achieves acceptable voice quality with 990 ms of latency for 5 million simulated users.

1 Introduction

Telecom providers retain call records which include the participants and duration of every call. This metadata is highly sensitive for most users [19] and is especially problematic for journalists who need to keep their sources confidential [10]. As a result, call records are targeted in large-scale attacks [22] and collected by intelligence agencies; the NSA collected 434 million call records of Americans in 2018 [23]. Even if telecoms stop retaining call records, an attacker can monitor the network to learn about voice calls happening in real-time.

Achieving high performance while protecting communication metadata is challenging against an adversary that can compromise servers and tamper with network traffic. In order to hide communication patterns, messages between all users must be processed in a synchronous batch, so as to give the adversary the appearance that any pair of users might be communicating. This processing either requires CPU-intensive cryptographic primitives such as PIR [3], which are trustless, or the use of semi-trusted servers whose job is to mix the messages without revealing the mixing to the adversary. However, if an adversary can compromise some of the servers, messages must be routed through enough servers to ensure the adversary does not control every one of them, which increases latency.

Prior systems like Herd [18] and Tor [7] can support voice calls, but make strong assumptions that certain servers are not compromised or that the adversary is not monitoring the entire network. Systems that provide stronger guarantees suffer from high latency [3, 14, 16, 26, 30]. For example, Karaoke [16] routes messages through 14 servers to ensure messages are mixed despite many servers being compromised. At each hop, each server performs a public-key operation for every incoming message, which results in 8 seconds of latency for 4 million users with 0.24 kbit/s of throughput for each user. Karaoke is the fastest of these systems, but its performance an order of magnitude away from the latency and bandwidth requirements of voice calls.

This paper presents Yodel, the first metadata-hiding system for voice communication that defends against an adversary that compromises the entire network and compromises many servers. Yodel hides metadata by operating a set of servers that form a mixnet to shuffle user messages. To address the costly public-key cryptographic operations that are typically required in a mixnet, Yodel uses symmetric-key circuits through the mixnet to relay messages between two users. Users set up circuits using public-key cryptography, but individual messages sent over circuits benefit from low-cost symmetric-key cryptography, enabling Yodel to achieve high performance.

Although circuits offer high performance, using them securely required Yodel to address two technical challenges. The first challenge lies in the fact that circuits are used for multiple messages. Since servers maintain shared keys with each user for the duration of a circuit, a server may be able to learn information about a user over time. For example, if a user is briefly disconnected from the network, a server might observe that no message arrived on a particular circuit, and infer that the circuit belongs to that user. Yodel's key insight is the idea of *self-healing circuits*, which rely on honest servers to ensure that circuit traffic is maintained despite network interruptions, such as a user's network going offline, or an active attack on any part of the network.

The second challenge lies in generating *cover traffic*, so that each user's traffic pattern is always the same, regardless of whether they are in a conversation or not. Suppose Alice wants to call Bob, so she sets up a circuit through the Yodel mixnet. She tells Bob to connect to a specific Yodel server and request messages for a specific circuit endpoint, and Bob does the same for Alice. This allows Bob to receive Alice's messages (and vice-versa), while the mixnet hides

Permission to make copies of this work is granted without fee provided that copies are not distributed for profit and that copies bear this notice.
SOSP '19, October 27–30, 2019, Huntsville, ON, Canada
© 2019 David Lazar, Yossi Gilad, Nickolai Zeldovich
ACM ISBN 978-1-4503-6873-5/19/10.
https://doi.org/10.1145/3341301.3359648

who is sending those messages. If Alice is not talking to anyone, she must still appear to perform the same steps, so as to prevent the adversary from determining if she's in a conversation or not. That means setting up a circuit, as if she is sending messages to someone, and requesting messages from some circuit endpoint, as if she is receiving messages from someone.

Yodel relies on an external metadata-private messaging system for users to establish calls (by telling each other about their circuits). Suppose that Alice tries to call Bob but doesn't hear back from him because the attacker tampered with the external messaging system. In order to not reveal whether she is communicating or not, Alice must request messages from *some* circuit endpoint as part of her cover traffic. She doesn't know the ID of Bob's circuit endpoint (or whether Bob is even online). But requesting messages from her own circuit endpoint is problematic, because Bob might have actually received Alice's call, and is also requesting messages from the same circuit endpoint on the same server. If that were to happen, an adversary with access to that server would conclude that Alice and Bob were trying to talk.

Yodel addresses this challenge using *guarded circuit exchange*, a simple protocol that ensures users always have a circuit they can safely connect to. The insight is to have each user establish two circuits: one as a fallback for cover traffic, and another as a circuit for talking with a buddy. In case of any message loss during dialing, each user can safely connect to *either* their cover traffic circuit or the buddy's circuit, without leaking any metadata to the adversary.

We implemented a prototype of Yodel in Go and ran it on 100 servers across Europe and North America to evaluate its performance. Our experimental results show that Yodel provides voice communication with 990 ms of latency from the time a user sends a message to the time their buddy receives it, while supporting 5 million simulated users. The 990 ms latency is close to the underlying network latency of sending the messages in synchronous batches across 15 hops, with a one-way delay of 45 ms between servers at each hop. Our security analysis shows that the probability that an adversary learns any metadata from Yodel is negligible when messages are sent over 15 hops, under the assumption that servers are honest with 80% probability.

Yodel's latency is above the ITU G.114 recommendation for voice calls (at most 400 ms) [11], and our prototype uses a low-bitrate vocoder [27], but we find that it provides acceptable voice quality. As anecdotal evidence that the quality is acceptable, we have communicated over Yodel several times in the course of preparing this paper.

Our contributions are the following:

- The design and implementation of Yodel, a low-latency metadata-private communication system that can support voice calls.

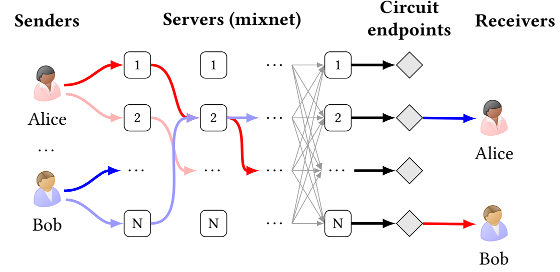

Figure 1. Overview of Yodel's components. Alice and Bob have created two circuits each. The faded arrows are backup circuits, created as part of Yodel's guarded circuit exchange. Alice and Bob are in a voice call, so they are connecting to each other's circuits, but the adversary doesn't know who created which circuit.

- The self-healing circuits and guarded circuit exchange mechanisms, which allow for efficient private communication through a mixnet.
- An analysis of Yodel's design and an experimental evaluation of its performance.

2 Overview

Figure 1 shows how users communicate through Yodel at a high level. Users send messages directly to a Yodel server which participates in a mix network with the other servers. The users choose a random sequence of servers to process each of their messages and onion encrypt their messages to ensure that messages follow their chosen paths. An established path through the network is called a *circuit*, and messages (e.g., voice packets) flow from users through circuits to their *endpoints*. Users receive messages by connecting to a circuit endpoint (i.e., connecting to a Yodel server and requesting messages for that endpoint ID). The endpoint ID is pseudorandom and reveals nothing about the sender.

Yodel's servers, labeled 1 through N in Figure 1, shuffle messages to hide which user is sending to which circuit endpoint. The servers shuffle messages in *layers*, indicated by the vertical groups, similar to a parallel mixnet [13, 16]. All paths in Yodel have the same number of layers, which is a system security parameter. At each layer, a server receives messages from all of the servers in the previous layer, decrypts the messages (which are onion-encrypted), shuffles them, and sends the messages to the servers on the next layer. To simplify Figure 1, the server-to-server communication is only shown for the next-to-last layer.

Yodel assumes that users know the servers' long-term public keys and that communicating users have established a shared secret out-of-band (e.g., using a metadata-private dialing system such as Alpenhorn [15]). Users use the shared secret to authenticate the circuit endpoint, which they communicate to their buddy through an external messaging system, and to encrypt their voice packets end-to-end. In Figure 1, Alice and Bob are in a voice call and have established two

circuits through Yodel, but each of them only connects to one circuit. Alice is sending messages to the circuit endpoint that Bob is connected to, and vice-versa. The adversary sees that Alice and Bob connect to the system, and knows to which circuit endpoints they are connected to. However, the mixnet hides which users are sending to which circuit endpoint, so the adversary cannot tell whether Bob is connected to Alice's circuit.

Communication through Yodel is divided into synchronous rounds and subrounds. In every round, each user establishes two new circuits, and each user also connects to some circuit endpoint. The two circuits are used for guarded circuit exchange: the user can send one to a conversation partner (if the user calls a buddy), and use the other one as a fallback for cover traffic (if they don't hear back from the buddy, or if they are not talking to anybody).

Each round has a fixed number of subrounds, with each circuit sending exactly one message per subround. Messages are encrypted with the keys of each hop in a circuit, in order, so that the message can be decrypted only if it correctly traverses the entire circuit. At every layer, each server collects all messages routed through it from the servers at the previous layer, decrypts each message with its corresponding circuit key, and sends them to their next hop, based on the pre-established circuit paths. Messages are always sent in batches, and the order of messages in a batch is determined at circuit setup time. This ensures message order cannot reveal any additional metadata during a subround.

If a server does not receive an incoming message on a circuit, it fills in random data in its place, and sends the random data to the next hop in the circuit. The random message is indistinguishable from a real message on the circuit, since messages are onion-encrypted at each hop. By filling in random messages in place of any missing messages, honest servers implement Yodel's self-healing circuits, ensuring that an adversary cannot trace the path of a circuit across an honest server by dropping or modifying messages. Yodel chooses circuit paths to be long enough to ensure an honest server is present on each path.

Although each user establishes two circuits, the user sends on just one of these circuits (the non-backup circuit); messages on the backup circuit are filled in with random bytes by honest servers, as if the messages were dropped. The two circuits are indistinguishable to the adversary, so sending messages on just one of them does not reveal any additional information. Similarly, only half of the circuit endpoints are connected to; for circuit endpoints with no connections, Yodel servers simply discard the messages.

2.1 Goals and threat model

Yodel has three goals: metadata privacy, high performance, and availability. Yodel provides privacy in two important dimensions. First, regardless of how many users are connected to the system, Yodel prevents an adversary from determining whether any pair of users are communicating or not, even if every other user is an adversarial Sybil. Second, by supporting a large number of users, Yodel makes it less suspicious for users to connect to Yodel in the first place [7]; otherwise, the mere use of Yodel may reveal critical metadata [8]. Yodel also tolerates some servers going down, as well as network outages, so that an attacker cannot easily take down the system with a denial-of-service attack.

Threat model. We design Yodel to resist attacks by a global adversary who has full control over the network and can tamper with messages traveling over any network link. Furthermore, we assume that the adversary controls some number of servers. To give an intuition for a possible parameter, studies on Tor suggest that less than 20% of the servers are malicious [21, 25, 31]. For most of this paper, we assume that each Yodel server has a 20% chance of being controlled by the adversary; however, this is just a parameter for Yodel, which influences the number of hops that messages must traverse. Finally, Yodel's design assumes that standard cryptographic constructs (e.g., private and public key cryptography and hash functions) are secure.

Security goal. Yodel's security goal is that the probability of an adversary learning any metadata about voice calls is negligible.[1] Specifically, we aim for the probability of an adversary learning anything to be 10^{-8} per round. We consider this to be a good security goal because we expect rounds to start every few minutes (so that a user need not wait more than a few minutes until they can establish a voice call in the next round). For example, starting a round every 5 minutes means it would take around 1000 years for the adversary to get lucky and learn a user's metadata for a single round. The reason our privacy guarantee is not stated in typical cryptographic strengths like "128 bits of security" is that it is primarily bounded by the number of rounds that an adversary can attack, rather than the computational resources available to the adversary.

Yodel does not hide which users are connected to the system. Instead, we aim to support many users so that it isn't suspicious for users to connect in the first place.

Performance goal. Yodel's performance goal is to support voice calls for many users. We aim to provide under one second of one-way latency for voice packets. Yodel also needs sufficient throughput to transmit audio between users, which is determined by the audio codec. Yodel targets the LPCNet vocoder [27], which is specialized for low-bandwidth speech transmission, and requires 1.6 kbit/s per user. We also evaluate Yodel with the standard Opus audio codec at 8 kbit/s per user. Finally, Yodel aims to support many users (e.g., millions running on 100 servers), and can scale to support more users by adding more servers.

[1] The probability is exponentially decreasing in the number of mixnet layers.

Envisioned deployment. To prevent an adversary from compromising a significant fraction of the servers, we envision that many organizations take part in running Yodel servers, across different administrative domains and government jurisdictions. Yodel's latency is dominated by the maximum latency between two servers in the system (because at each hop, servers wait to receive messages from all other servers). Thus, servers should be relatively close to minimize this latency. Our evaluation (§6) uses servers on the east coast of the United States and distributed across countries in Europe, with a maximum one-way latency of 45 ms between servers. In this setup, the lower bound on Yodel's one-way latency, assuming circuits are 15 hops long, is 45 ms $\times$ 15 hops = 675 ms. Another possibility with more political diversity but similar proximity is to deploy servers in Europe, Israel, and Russia.

We envision that the policy for adding servers to Yodel is stricter than Tor, which allows any server to automatically join the network. In Yodel, all servers participate in all layers of the mixnet, so adding a new server immediately impacts the performance of all users in the system (for better or worse). One possible policy for Yodel is to require new servers to be manually approved by an independent organization.

Availability. Yodel is resilient to some servers being down at the start of a round (up to 2%), and to temporary large-scale server or network outages that occur during a round. No matter how many servers are down, Yodel maintains its privacy guarantee. However, users who established circuits through a failing server will not be able to communicate messages to their partners. The external messaging system that Yodel uses to exchange circuit information should also be resilient to faults to ensure the availability of communication end-to-end.

3 Design

Calls through Yodel operate in rounds, which are kicked off by one of Yodel's servers that acts as a coordinator and notifies the other servers about the new round. Clients connect to one of the Yodel servers to receive notifications about new rounds; this server is known as the client's *entry* server. Round numbers must increase with every announcement so that the coordinator cannot announce the same round multiple times. The coordinator is untrusted, and if the coordinator goes down the servers can elect a new server to act as coordinator (in practice, the online server with the smallest long-term public key is the coordinator by fiat).

We explain Yodel's design using pseudocode for the Yodel client, shown in Figure 2. Every time the servers announce a new round, the `client_round` function in Figure 2 is called with the new round number, the public keys of the Yodel servers, and the user's call buddy. Specifically, the `onion_keys` parameter is an array with a unique public key for each of

```
def client_round(round, onion_keys, buddy):
  ### Phase 1: Circuit Setup
  onion1, circuit1 = rand_circuit(round, onion_keys)
  onion2, circuit2 = rand_circuit(round, onion_keys)

  r1, r2 = send_setup_onions([onion1, onion2])
  h1 = onion_decrypt_aes(circuit1.keys, r1)
  h2 = onion_decrypt_aes(circuit2.keys, r2)
  if h1 != hash(circuit1.endpoint) or
     h2 != hash(circuit2.endpoint):
    raise Exception("hash mismatch; aborting round")

  ### Phase 2: Noise Verification
  amounts, sigs = recv_noise_signatures()
  noise = 0
  for i in range(servers):
    if verify(sig[i], servers[i].signing_key):
      noise += amounts[i]
  if noise < required_noise:
    raise Exception("insufficient noise present")

  ### Phase 3: Circuit Exchange
  buddy_endpoint = exchange_circuit(buddy, circuit1.endpoint)
  if buddy_endpoint is None:
    buddy = self
    buddy_endpoint = circuit2.endpoint

  conn = connect_circuit(buddy_endpoint)

  ### Phase 4: Circuit Messaging
  def read_loop():
    while data := conn.read():
      # Note: don't need to onion_decrypt_aes here.
      msg = decrypt_aes(buddy.secret, data)
      play_voice_packet(msg)
  spawn_thread(read_loop)

  while r := recv_subround_announcement():
    msg = encrypt_aes(buddy.secret, get_voice_packet())
    onion = onion_encrypt_aes(circuit1.keys, msg)
    send_voice_onion(r.subround, onion)
```

Figure 2. Pseudocode for the Yodel client. Several details (e.g., MACs, nonces, and key rotation) are omitted for clarity.

Yodel's N servers, freshly generated for the round.[2] The buddy parameter is an object that contains information about the user's call partner, including a shared secret that they established out-of-band. If the user doesn't have a call partner for some round, then the `self` object is used for the buddy parameter (so idle users effectively chat with themselves).

Every round is divided into four phases, as indicated in Figure 2. Phases 1–3 enable clients to build and connect to circuits securely. Then clients spend most of the round in phase 4, exchanging voice packets with their call buddy. In the following sections, we explain each of these phases in detail, both from the client's perspective and the server's perspective.

[2]The newly generated keys are signed by the servers' long-term signing keys, but we omit the signing and verification steps from the pseudocode.

```
def rand_circuit(round, onion_keys):
  endpoint = rand.bytes(32)
  path = [rand.choice(onion_keys) for i in range(nlayers)]
  onion, aes_keys = onion_encrypt(path, endpoint)
  return onion, Circuit(aes_keys, endpoint)

def onion_encrypt(path_public_keys, msg):
  keys = []
  onion = msg
  for srv in reversed(path_public_keys):
    pub, priv = generate_key_pair()
    shared_key = diffie_hellman(priv, srv.public_key)
    ctxt = encrypt_aes(shared_key, srv.next_hop_idx + onion)
    # Note: ciphertext expansion due to public key and MAC.
    onion = pub + ctxt + MAC(shared_key, ctxt)
    keys = [shared_key] + keys
  return onion, keys

def onion_encrypt_aes(path_aes_keys, msg):
  onion = msg
  for key in reversed(path_aes_keys):
    # Note: no ciphertext expansion!
    onion = encrypt_aes(key, onion)
```

Figure 3. Pseudocode for creating circuits and onions; used by clients and servers. We use AES in the pseudocode for concreteness, but Yodel is not tied to a particular cipher.

3.1 Circuit setup

During every circuit setup phase (phase 1 in Figure 2), clients create two circuits through Yodel's mix network. The client uses one circuit for sending messages to the user's call buddy, and the other as a fallback in case the circuit exchange step fails (as we explain later).

Clients create circuits by sending onion-encrypted messages to the mixnet. To set up a circuit, the client calls the `rand_circuit` function in Figure 3 which selects a random 256-bit identifier for the circuit endpoint, and then selects one of Yodel's servers at random for every layer through the mixnet. The client then creates a *circuit setup onion* that consists of the endpoint ID, repeatedly encrypted using the public key of each randomly selected hop in the circuit.

The `onion_encrypt` function (Figure 3) creates the circuit setup onion by adding layers of encryption in reverse order of the circuit's path, starting with the endpoint ID. At each layer, the client generates an ephemeral key pair which is used to derive a shared key using Diffie-Hellman. The onion is encrypted with the shared key along with an index that identifies onion's next hop. Finally, the ephemeral public key is appended to the onion so that the server can derive the same shared key and decrypt one layer of the onion. The `onion_encrypt` function also returns the shared keys used to encrypt the onion, as those will be used to encrypt voice packets during the circuit messaging phase. The client then sends the circuit setup onions through the mixnet (by the calling `send_setup_onions` in Figure 2).

Server-side processing Figure 4 shows the pseudocode for how a server handles circuit setup onions at a particular layer. Each server receives as inputs the messages routed through it from the servers on the previous layer (the servers on the first layer collect messages from users). The servers discard duplicate messages, which is essential for security. If an attacker manages to duplicate a user's circuit setup onion, it will result in two circuits with the same endpoint—a pattern that links the user to their circuit's endpoint. Since Yodel's onion encryption scheme during circuit setup is non-malleable, removing duplicates is just a matter of dropping identical messages.

```
def process_circuit_setup(round, layer, inputs):
  inputs = dedup(inputs)
  priv = srv.get_private_key(round)

  for i in range(inputs):
    keys[i] = diffie_hellman(priv, inputs[i].public_key)
    msgs[i] = decrypt_aes(keys[i], inputs[i].msg)

  if layer < nlayers-1:
    shuffle = rand.permutation(len(msgs))
    shuffle.apply(msgs)

    hops = [msg.next_hop for msg in msgs]
    srv.circuit_state[(round,layer)] = (keys, shuffle, hops)

    replies = distribute_setup_onions(layer+1, msgs, hops)
    shuffle.invert(replies)
  else: # Last layer in the mixnet:
    endpoints = msgs
    srv.circuit_state[(round,layer)] = (keys, endpoints)
    replies = [hash(endpoint) for endpoint in endpoints]

  for i in range(replies):
    replies[i] = encrypt_aes(keys[i], replies[i])
  # Replies are sent to previous layer, or users.
  return replies
```

Figure 4. Server pseudocode for circuit setup. The noise generation and verification steps (§3.2) happen before and after this code runs, and are not shown here.

For each input message, the server peels one layer of onion encryption by computing the shared key using Diffie-Hellman (with its private key and the onion's public key) and using it to decrypt the message. If the server is processing a non-final layer, it mixes the decrypted messages by generating a random permutation and then shuffling the messages according to that permutation. This step is what prevents the adversary from connecting senders to receivers, assuming the permutation stays hidden.

The servers implement persistent circuits by storing the symmetric key and next hop of each onion and the shuffle permutation for each layer in the `circuit_state` map. When messages are sent through circuits (in subrounds), the server decrypts each input using its corresponding symmetric key (i.e., `inputs[i]` is decrypted with `keys[i]`), applies the saved permutation, and sends each message to the next hop on its circuit. By using the same permutation, servers can identify

messages belonging to the same circuit across subrounds, and use the corresponding symmetric keys to decrypt them.

Continuing with the code in Figure 4, the server relays messages to their next hop on the next layer by calling `distribute_setup_onions`. This call blocks until the next layer returns a reply message for each onion. On the last layer, servers decrypt the circuit setup onions to learn the random 256-bit circuit endpoints corresponding to the circuits. The servers save the endpoints so that users can later connect to the corresponding circuits and receive messages. The last layer replies to each circuit setup onion with a cryptographic hash of its endpoint, called a *receipt*, which enables users to verify that their circuits were created correctly.

The replies flow through the mixnet in reverse, back towards the users. Each server on the reverse path waits for the replies from the following layer, then applies the inverse of the permutation it used for shuffling messages on the forward path. To prevent an adversary from correlating messages on the reverse path, the server encrypts the replies with the shared keys from the forward path. Eventually, a symmetrically encrypted onion message carrying the hash of the circuit endpoint reaches the client. The client decrypts the onion, and checks that the circuit was set up correctly by comparing the hash of the endpoint ID it had selected against the hash specified in the returned message (as shown at the end of phase 1 in Figure 2). If the hashes match, then the client is guaranteed that the circuit setup onion traversed all of the servers on its chosen path. A client that fails to establish two circuits will not proceed with the round. This completes the circuit setup phase.

3.2 Noise generation

Yodel's privacy guarantee relies on unlinking the user who creates a circuit from the circuit's endpoint. In §4 we show that by shuffling messages at honest mix servers, Yodel prevents (with overwhelming probability) an attacker from learning whether Alice is connecting to Bob's circuit or her own (i.e., whether Alice is chatting with Bob or she is idle).

In Yodel's topology, users create circuits that take independent routes through the mixnet. This approach distributes the load over all available servers, which allows Yodel to reach its performance goal but also introduces risk. If two users, Alice and Bob, set up non-intersecting circuits, then an attacker that discards circuit setup messages from all other users could trace Alice's circuit to its endpoint and detect whether Bob is connecting to it. The more servers Yodel has, the higher the chance that Alice and Bob pick non-intersecting paths for their circuits. Yodel addresses this issue by having its servers create *noise circuits* (similar to noise messages in Karaoke [16]) during the circuit setup phase and ensuring that these circuits are established before users begin connecting to each other's circuits. The noise circuits ensure that every user's circuit intersects with some circuits whose routes the attacker does not know. Much like regular clients, servers select a random path through the mixnet for their noise circuits and verify that their circuits were established by checking the receipt (as described in §3.1). Our analysis computes the number of noise circuits that every server needs to create to ensure Yodel's security goals (§4).

3.3 Noise verification

One challenge in relying on noise circuits is that an attacker might drop the circuit setup messages to eliminate the noise in the system. Yodel prevents this attack by having servers announce if their noise circuits have been successfully created, before users attempt to connect to any circuits. Concretely, after a server creates and verifies the receipts of its noise circuits for a round, it broadcasts a *noise signature* to all other servers indicating how much of its noise is present for the round. Each user's entry server aggregates these signatures[3] and forwards them to the user's client.

The client verifies that servers generated enough noise before proceeding with the round. First, it receives the noise signatures from its entry server and the amount of noise that each server vouched for, as shown in phase 2 of Figure 2. The amount of noise that a server vouches for is dynamic to handle faulty servers, as we describe next. The client determines how much total noise is present in the round by adding together the per-server noise amounts that have valid signatures. If the total noise is over the threshold for privacy, which is a system parameter (`required_noise` in the code), the client continues to the next phase of the round, otherwise it aborts the round.

Handling faulty servers. When servers are down, there might not be enough noise for clients to proceed with the round. Yodel deals with this by having online servers generate extra noise when servers go down. One limitation of Yodel (discussed further in §7) is that a high percentage of messages get lost once a few servers go down, due to requiring messages to traverse many hops. For example, if 2% of the servers go down, 20 hops results in up to $2\% \times 20 = 40\%$ message loss. However, if each server generates 1.7× more noise when 2% of the servers go down, then noise verification can still succeed. In this case, verification succeeds if each server receives 60% (budgeting for 40% loss) of the receipts for their noise circuits (since $0.6 \times 1.7 > 1$).

3.4 Guarded circuit exchange

After clients establish circuits and verify that servers have generated sufficient noise for the round, they need to choose a circuit endpoint to connect to for the remainder of the round. To start a voice call, clients exchange circuit endpoints through an external metadata-private messaging system by calling `exchange_circuit` in phase 3 of Figure 2. In the case that Alice is calling Bob, she sends Bob the endpoint ID of one of her circuits (`circuit1`) through the external messaging

[3] An aggregatable signature scheme like BLS [5] could save bandwidth.

system. If the exchange succeeds, then the function returns Bob's response (an endpoint that he created). However, the adversary can block messages over the external system.

Yodel's adversary model allows the attacker to discard any message sent on the network. Therefore, the attacker can discard Alice's message and prevent her from notifying Bob about her circuit's endpoint. If a client does not receive a circuit endpoint from their buddy (i.e., `buddy_endpoint` is `None` in phase 3 of Figure 2), then the client connects to the backup circuit it had established (`circuit2`). Users never share the endpoint of their backup circuit with anyone, ensuring no other user will connect to that circuit.

This backup circuit is crucial since Alice can never know whether Bob received her circuit endpoint and vice-versa—this is the Two Generals problem [2, 9]. However, Alice needs to connect to *some* circuit to ensure her traffic patterns are the same in every round. Since she can never be sure about the state of `circuit1`, she connects to `circuit2` if she does not receive a circuit endpoint from Bob. If Alice is idle (i.e., not calling anyone), then her client still invokes the external messaging service with a message to herself as a form of cover traffic.

After choosing which circuit to connect to, the client calls `connect_circuit` to start receiving messages from that circuit. The circuit endpoints contain information about which server on the last layer is hosting that circuit, so that the client knows which server to connect to.

Yodel's end-to-end guarantees are only as strong as the guarantees offered by the external system used to exchange circuits. The external system needs to have strong security properties, but also needs good enough performance so that users can establish calls quickly. For example, Pung [3] offer strong security, but it could take several minutes to establish a call. Alternatively, Karaoke [16] provides a weaker guarantee, but its lower message latency (e.g., 8 seconds for 4 million users) would allow users to establish calls more quickly.

3.5 Circuit messaging and self-healing circuits

Once clients have set up circuits and exchanged their endpoints, users can start exchanging messages. Yodel divides every round into a fixed number of subrounds (e.g., 1,000), which the coordinator kicks off at fixed intervals and entry servers announce to their clients.

The client pseudocode for circuit messaging is shown in phase 4 of Figure 2. In every subround, the client sends a fixed-size message to their non-backup circuit (`circuit1`), intended for the user's call buddy. The content of the message (e.g., a voice packet) is encrypted end-to-end with a key known only to the user and their call buddy. The encrypted content is then onion-encrypted using the symmetric keys on the circuit's path, which the sender established during circuit setup. The client sends the onion to the first hop on its circuit by calling `send_voice_onion`.

```
def process_subround(round, layer, subround, inputs):
  st = srv.circuit_state[(round,layer)]
  for i in range(inputs):
    if inputs[i] == None:
      # Heal the missing input.
      outputs[i] = rand.bytes(subround_msg_size)
    else:
      outputs[i] = decrypt_aes(st.keys[i], inputs[i])

  if layer < nlayers-1:
    st.shuffle.apply(outputs)
    # No replies since circuits are unidirectional.
    distribute_voice_onions(layer+1, outputs, st.hops)
  else:
    # Last layer delivers messages to users.
    for i in range(st.endpoints):
      if u := connected_user(st.endpoints[i]):
        send_msg(u, outputs[i])
```

Figure 5. Server pseudocode for circuit messaging.

In a separate thread, the client receives one message every subround from their buddy's circuit endpoint. The message is decrypted with the buddy's shared secret and the resulting audio data is sent to the user's speaker. The end-to-end encryption between users can optionally include authentication. However, it is crucial that the user not react to authentication failures (e.g., by going offline), as this would undermine Yodel's self-healing, which we describe next.

Server-side processing. Figure 5 shows the pseudocode for processing a subround on a particular layer. On every layer, the server receives as input the messages from the previous layer in batches. The input message to each circuit is determined by the position of the message in the batch. If the input to a circuit is present, the server removes a layer of encryption from the message using the circuit's key, which was established during the circuit setup phase. If some circuit is missing a message, the server *heals* the circuit by generating a random message in its place, which defeats active attacks.

A critical challenge that Yodel handles is that the adversary might drop messages in an attempt to correlate traffic between subrounds. For example, if whenever the attacker drops Alice's message, Bob does not receive a message from the endpoint he is connected to, then they must be talking. Yodel addresses this challenge with the idea of self-healing circuits, illustrated in Figure 6.

To ensure that missing messages do not create an observable pattern, the server that detects the loss creates another message in its place. The server replaces the missing message with random bytes, as shown in Figure 5. Importantly, during subrounds, messages are onion-encrypted with the symmetric circuit keys, but the messages are not authenticated (see the `onion_encrypt_aes` function in Figure 3), so that a random string is indistinguishable from the original message to everyone except for the sender and their buddy.

After decrypting the input messages and healing any missing inputs, the server applies the shuffle that was generated

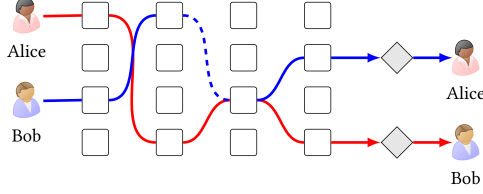

Figure 6. Self-healing. The dashed line denotes a message that the attacker drops on the blue circuit. The honest server on the third layer fills a message in its place, which ensures that the attacker cannot distinguish red and blue messages after the third layer.

during circuit setup and sends the results to the next layer. Using the same shuffle ensures the next layer will be able to map input messages to the correct circuits, so that servers apply the right circuit keys. A malicious server could shuffle messages under a different permutation so that the next server applies the wrong keys, but this would only cause the user's call buddy to fail in decrypting those messages (and thus discard them). It does not benefit the attacker since Yodel messages are not authenticated between hops, and any message looks equally plausible.

The last layer hosts the circuit endpoints, and users connect directly to a server to request messages for an endpoint. If a circuit endpoint has no connected user, the server discards messages that arrive on that endpoint.

4 Analysis

Our privacy analysis follows the structure of the client pseudocode from Figure 2, where privacy means hiding the user's `buddy`. The first two phases, circuit setup and noise verification, are independent of the user's buddy, and thus leak no information about who the user is communicating with. If any errors arise at this point, the client will stop participating in this round, and leak no further information about `buddy`. The third phase involves the external messaging system for circuit exchange, whose privacy is outside of the scope of our analysis; we assume it provides sufficient guarantees. The third phase also involves the client connecting to a specific circuit endpoint. §4.1 argues that this leaks no information about `buddy`, because the adversary cannot determine which user established a given circuit endpoint.

Once users set up their circuits, the fourth phase involves sending messages over these circuits. The attacker observes the same communication pattern in every subround: users send messages to the same servers, every inter-server link carries the same number of messages, and users receive one message from the same endpoint. (Yodel's self-healing ensures that the attacker observes the same pattern even if the attacker discards messages.) Therefore, exchanging many messages does not allow the attacker to learn any more information about the conversation's metadata than just a single exchange, as in the circuit setup phase.

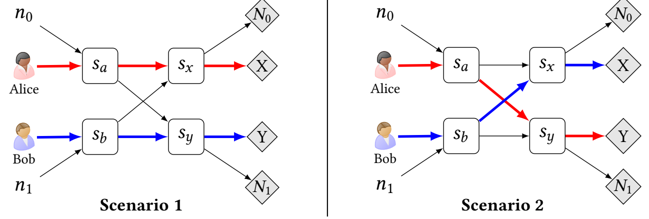

Figure 7. Yodel's privacy guarantee: an adversary cannot determine whether Alice's message goes to circuit endpoint X and Bob's message to Y or vice-versa (i.e., scenarios 1 and 2 are equally likely given the adversary's observations). Lines represent links with malicious intermediary servers that an adversary can track. Servers s_a, s_b, s_x, and s_y are honest; n_0 and n_1 are noise messages.

We omit many of the details in this section; a companion analysis [17] provides a more detailed treatment of Yodel's privacy guarantees.

4.1 Circuit indistinguishability

Yodel's privacy stems from the adversary's inability to correlate the start and end points of a user's circuit. To make this more precise, we introduce the notion of *peering circuits*, as illustrated in Figure 7. Two circuits peer if both circuits route through at least two honest servers, and the leftmost honest server on each circuit's path (s_a or s_b in Figure 7) is to the left of the rightmost honest server on the other circuit's path (either s_y or s_x in Figure 7, respectively). Server s is "left" of s' if s appears on the route in an earlier layer than s'. With sufficiently long routes, we can ensure that circuits peer with high probability, as we analyze in §4.2.

Theorem 1. Conversation privacy. For any two peering circuits created by honest clients, Alice and Bob, with endpoints x and y, the following holds:

$$|\Pr[\text{Alice} \to x \wedge \text{Bob} \to y \mid O] \\ - \Pr[\text{Alice} \to y \wedge \text{Bob} \to x \mid O]| \le \eta,$$

where user $\to x$ means that user created the circuit with endpoint x, and O denotes the attacker's observations from all network links and compromised servers. The probability is taken over the coin tosses in the selections of the circuits' paths and the cryptographic primitives that Yodel uses. η is a negligible function in the number of noise circuits and the security parameters of Yodel's cryptographic primitives.

Proof. Observe Figure 7. In this figure two users, Alice and Bob, are sending messages through the mixnet. The messages n_0 and n_1 are noise messages and happen to coincide with the messages from Alice and Bob at the honest servers s_a and s_b respectively. For simplicity, assume that the attacker has discarded all other messages, so the attacker sees precisely one message on every link. Server s_a shuffles Alice's message with n_0; after the server peels its layer of the onion encryption and uncovers the next layer of the onions, the two messages

appear indistinguishable (by the cryptographic merits of the encryption scheme). It is therefore equally likely (given the attacker's observations) that Alice's message travels to s_x and n_0 travels to s_y or vice-versa. Similarly, Bob's message is equally likely to be at s_x and n_1 at s_y or vice-versa.

Since no user connects to N_0 or N_1, the attacker can infer that these endpoints belong to noise circuits, so Alice and Bob must route their messages to endpoints X and Y. The attacker cannot distinguish whether n_0 arrives at endpoint N_0 and n_1 arrives at N_1 or vice versa, so there are two equally likely cases, corresponding to the two scenarios in Figure 7. Scenario 1: Alice's message travels to s_x and then to endpoint X, and Bob's message to s_y and then to endpoint Y (and n_0 reaches endpoint N_1 and n_1 reaches endpoint N_0). Scenario 2: Alice's message travels to Y, Bob's message travels to X, and n_0/n_1 travel to N_0/N_1.

To complete the argument, we need to show that a noise circuit that routes from s_a to s_x and another circuit routing from s_b to s_y (or alternatively, routes from s_a to s_y and s_b to s_x) exist with overwhelming probability. The probability that a noise circuit routes through two particular servers is $\frac{1}{N^2}$, so the probability that all m noise circuits do not route through these servers is $(1 - \frac{1}{N^2})^m$. Using the union bound we find that the probability that there is not a pair of circuits where each circuit route through the servers above is less than $2(1 - \frac{1}{N^2})^m$, which (for a fixed N) approaches to 0 as m increases. □

Informally, Theorem 1 means that the attacker cannot distinguish which of the two peering circuits was created by Alice and which by Bob. This is important since the attacker can see the endpoint that a user connects to (i.e., if last server on the circuit's path is corrupt). If an adversary could determine that Alice is connecting to Bob's circuit endpoint, the adversary would learn that they are communicating.

The companion analysis [17] shows how Theorem 1 can be extended to any number of pairs of peering circuits to provide *group privacy*. The group privacy guarantee allows any set of communicating users to claim they were idle and any set of pairs of idle users to claim they were communicating, which is stronger than the two-user guarantee of Theorem 1 and prior systems [16, 26, 30]. For example, if an adversary wishes to learn whether any of an organization's employees communicate with some journalists, then the two-user guarantee is insufficient. In contrast, group privacy applies to any set of pairs of peering circuits so it can protect any group of users.

4.2 Security parameters

To achieve meaningful protections with Theorem 1 (and the stronger group privacy guarantee), circuits must peer and there must be sufficient noise in the system. This section analyzes the parameters that enable Yodel to meet these conditions with high probability.

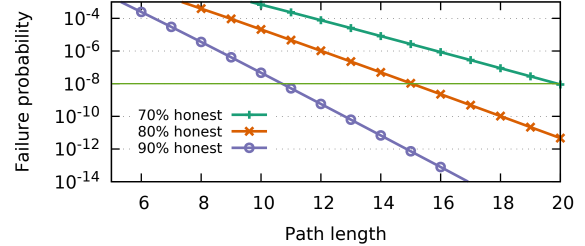

Figure 8. Probability for two circuits *not* peering as a function of the path length for varying trust assumptions. Our experiments target a failure probability of 10^{-8}.

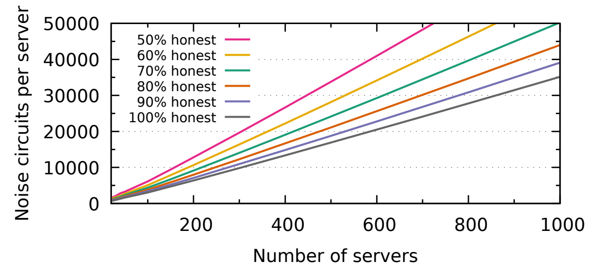

Figure 9. Number of noise circuits per server needed to guarantee *group privacy* with high probability $(1 - 10^{-8})$, for circuits with up to 20 hops. The number of noise circuits is dependent on the number of servers and the probability of each server being a honest.

The probability that two circuits peer increases with the number of hops in the circuits. Our companion analysis [17] shows how to compute this probability, and Figure 8 gives the results for various trust assumptions. The results show that if servers are honest with 80% probability, then two circuits with 15 hops peer with probability $1 - 10^{-8}$. Our analysis also shows that Yodel's path length is close to optimal for its parallel mixnet topology (which is also used by prior systems [13, 16, 24]).

To guarantee group privacy, servers must also create sufficient noise circuits during the circuit setup phase. Figure 9 presents the number of noise circuits needed for different deployment sizes and trust assumptions. We find that the amount of noise that each server should generate grows proportionally to the number of servers. This growth seems unavoidable with Yodel's topology, since all servers (together) need to generate noise proportional to the number of inter-server links (which is quadratic in the number of servers).

5 Implementation

The Yodel prototype is around 10,000 lines of Go code and is distributed as part of Vuvuzela at https://github.com/vuvuzela. It uses the NaCl box primitive [4] to onion encrypt circuit setup messages and native AES instructions to onion encrypt voice frames during circuit messaging. During circuit setup, servers send each other batches of onions over TCP using the

gRPC library. The circuit messaging phase switches to UDP to send batches of AES-encrypted onions between servers.

We opt for UDP in circuit messaging because with TCP a single packet drop anywhere in the network stalls the entire subround,[4] increasing latency for all users. This is problematic for voice calls, where a moment of silence is preferable to hundreds of milliseconds of extra lag. Self-healing circuits ensure that dropped message do not impact security, which enables us to use UDP to avoid retransmission delays.

During circuit messaging, after a server decrypts all the onions in a layer, our Go code writes out the entire batch of onions to the UDP socket at once using the `sendmmsg` system call. This bursty behavior, combined with the synchronous nature of subrounds, yields significant UDP packet loss without rate limiting. Our implementation relies on the `htb` qdisc in Linux for rate limiting. At deployment time, each Yodel server creates an `htb` qdisc for every other server in the network and the server's total outgoing bandwidth is evenly allocated among the qdiscs. For example, in a deployment with 100 servers, a server with a 10 Gbit/s link creates 100 qdiscs, each with a max rate of 100 Mbit/s, and maps each server connection to one of those qdiscs. This enables Yodel to achieve a loss rate of less than 0.1% for all data points in our experiments (§6).

Our implementation supports two audio codecs: LPCNet [27] and Opus [28, 29]. The choice of codec impacts Yodel's message size and subround frequency, which are fixed at deployment time. In LPCNet, an audio frame is 40 ms and compresses to 8 bytes, so Yodel uses 64 bytes to encode 7 frames every subround (the remaining 8 bytes are used for a keyed checksum to detect loss in the presence of self-healing). With this encoding, Yodel runs a subround every 280 ms to achieve continuous playback, resulting in 1.6 kbit/s of throughput per user. In Opus, each audio frame is 60 ms and compresses to 60 bytes, so we fit 4 audio frames into a 256-byte message every subround. In this mode, we run subrounds every 240 ms, which results in 8 kbit/s of throughput per user.

6 Evaluation

We experimentally answer the following questions:

- Can Yodel achieve its latency and bandwidth targets to support voice calls for a large number of users?
- Can Yodel scale to more users by adding more servers?
- How do trust assumptions impact Yodel's performance?
- What are the major costs of running a Yodel server?
- Does Yodel provide acceptable voice quality?

We simulated a realistic deployment of Yodel by running it over the internet with servers in different countries. The Yodel servers ran on Amazon EC2, evenly distributed among five data centers in different countries: Virginia (us-east-1), Ireland (eu-west-1), London (eu-west-2), Paris (eu-west-3), and Frankfurt (eu-central-1). We chose these regions to minimize the network latency between servers while maximizing the number of independent "trust zones" that the servers operate in. The links between Virginia and Frankfurt had the highest latency, with a weekly average of 90ms (round-trip); the latencies between servers in Europe ranged from 15ms to 40ms [1].

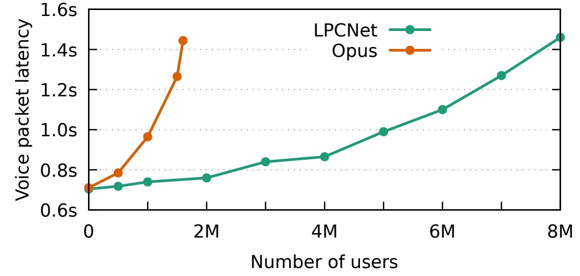

Figure 10. One-way latency for voice packets with a varying number of users and 100 Yodel servers.

Each Yodel server ran on a `c5.9xlarge` EC2 instance (Intel Xeon 3.0 GHz CPUs with 36 cores, 72 GB of memory, and a 10 Gbit/s link). On each server, we dedicated 30 cores to running circuit messaging subrounds for the current round, and the remaining 6 cores to running circuit setup (including noise generation and verification) for the next round.

We simulated millions of users by having servers create extra circuits during circuit setup (2 per simulated user). Even though users don't connect to these circuits, each circuit corresponds to the load of a real voice call. However, we exclude the cost of generating the extra circuit setup onions (which would normally be done by clients) in our results.

Two real users ran the voice call client at their homes in Boston, which were used to measure end-to-end latency and throughput. The maximum round-trip latency from both users to a Yodel server was 90 ms. The real users ran the Alpenhorn [15] dialing protocol to agree on a shared secret out-of-band. Lastly, the Karaoke [16] chat system was used for circuit exchange, but it ran on separate servers.

All experiments, except for those in §6.3, simulated the assumption that servers are honest with 80% probability, which required that users' messages pass through 15 hops to achieve Yodel's security goal. The experiments also targeted a failure probability of 10^{-8}, which means that each server generated around 3700 noise circuits in every round when Yodel was deployed with 100 servers. Finally, the relative standard deviation of each data point is ≤ 6%.

6.1 Yodel achieves sub-second latency

Figure 10 shows the results of measuring the end-to-end latency of voice packets through Yodel as we varied the number of users connecting to 100 servers. The results show

[4] A server must wait for the packet to be retransmitted before the kernel gives it the rest of the messages in the batch so it can perform the shuffle.

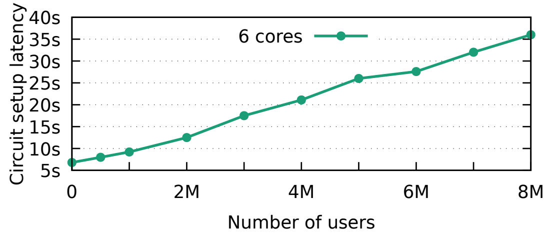

Figure 11. End-to-end latency of circuit setup with 100 Yodel servers and a varying number of users.

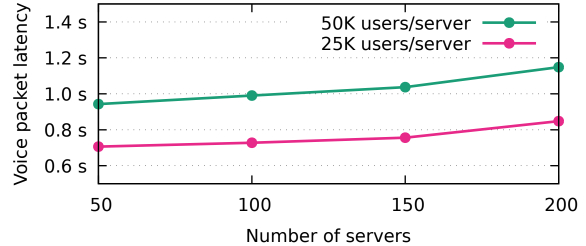

Figure 12. One-way latency for voice packets with a fixed number of users and a varying number of servers. The right-most points correspond to 5M and 10M users.

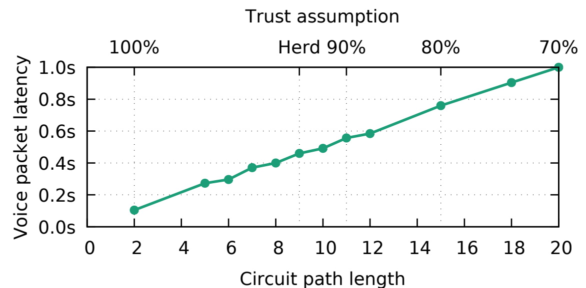

Figure 13. One-way latency for voice packets with 100 servers, 2 million users, and a varying path length. The top of the graph is annotated with the trust assumptions about server honesty that translate into a given path length.

that 100 Yodel servers can support voice calls for 5 million users with under 1 second of one-way latency. Beyond 5 million users, the latency grows to 1.4 seconds, which we consider too high for voice calls—Yodel would need more servers to support voice calls for that user load. Beyond 8 million users, packet loss between servers made it difficult to sustain the end-to-end throughput needed for a voice call with LPCNet.

We also evaluated Yodel using the standard Opus audio codec that is used in most VoIP applications. Opus with the lowest quality settings uses 5× more bandwidth than LPC-Net, hence Yodel is unable to support as many users in this configuration, as shown in Figure 10. With Opus, 100 servers can support 1 million voice calls with 965 ms of latency.

Yodel provides a seamless audio transition between rounds by running circuit setup for the next round in the background of the current round. To avoid interfering with circuit messaging, we use only a few cores on each server to setup circuits. Figure 11 shows the time it took to run circuit setup on 6 of the 36 cores in our VMs, with a varying number of users. The results show that Yodel is able to complete circuit setup in under 40 seconds with 8 million users. Since rounds start every five minutes, circuit setup finishes with plenty of time to spare, enabling Yodel to provide continuous audio playback to users over long conversations.

6.2 Yodel scales by adding servers

Yodel is designed to support more users by adding a proportional number of servers. To evaluate if this is the case, we measured the end-to-end latency of Yodel with a varying number of servers and a proportional number of users. We ran two experiments. In the first experiment, we added 25K users to the system every time we added a server to the network. In the second, we added 50K users per server.

Figure 12 shows the results, which indicate that Yodel's latency goes up slightly as the system scales to more servers and users. The reason is that the number of noise circuits required for security is dependent on the number of servers in the system (as explained in §4). For example, with 200 servers each server is required to create 4× as many noise circuits as in a configuration with 50 servers, resulting in increased latency. Nevertheless, 200 servers can support 5 million users with 848 ms of latency.

6.3 Stronger trust assumptions improve latency

Our baseline assumption is that servers are honest with a probability of 80%, which requires paths to consist of 15 hops. Other trust assumptions translate into different path lengths, as shown in Figure 8. Figure 13 shows the effect of path length on Yodel's latency. Increasing the path length causes a linear increase in latency, but enables Yodel to tolerate a higher chance of a server being compromised. If the adversary is assumed to control the network but none of the servers (100% honest servers), Yodel requires paths of length 2, which translates into a latency of around 100 ms.

Figure 8 also compares Yodel with Herd [18], the only other system that specifically aims to protect metadata for voice calls. Herd assumes that the first server on a user's path is honest. If we make a similar assumption, then Yodel's mixnet needs 9 layers to meet our security goal, which results in around 450 ms of latency for 2 million users and 100 servers. Herd with 10 million users and 1000 servers achieves 200 ms for users in North America, but its performance comes with a weaker privacy guarantee, as we discuss in §8.

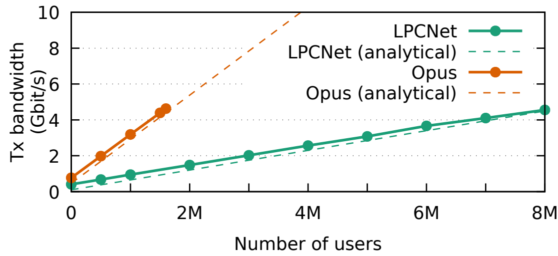

Figure 14. Average transmit bandwidth per server, with 100 servers and a varying number of users.

6.4 Server costs are dominated by bandwidth

The most significant cost in running a Yodel server is the bandwidth due to millions of ongoing voice calls (2 for each user). Figure 14 shows the average transmit bandwidth usage of a single server as we varied the number of users in the system. The results show that with 5 million users using LPCNet, a single Yodel server sends at a rate of 3 Gbit/s on average. Our implementation could not sustain over 4.6 Gbit/s across the internet without significant packet loss, so 100 servers could only support 1.75 million users using Opus.

The dashed lines in Figure 14 show the computed bandwidth a Yodel server would need to send just audio (two calls per user divided evenly among the servers), without any processing or security guarantees. The results show that Yodel's bandwidth usage is nearly optimal in this regard. The reason is that circuit messaging has no bandwidth overhead due to onion-encryption and only a small amount of overhead due to checksums between users.

Although Yodel's bandwidth costs are high, we believe it is possible to deploy Yodel by charging users for bandwidth. In 2016, the cost of a 10 Gbit/s link from the U.S. to Europe was around $4000/month [6], which suggests that Yodel could charge users less than $1/year to cover bandwidth costs.

6.5 Yodel provides acceptable voice quality

We had several productive conversations over Yodel using LPCNet and 5 million simulated users. The latency made it clunky to interject in the middle of someone's monologue, but otherwise the human-to-human information exchange was significantly higher than if we had been texting. We recorded a short conversation over Yodel running in this configuration, available at vuvuzela.io/yodel/audio-samples.

We found that voice quality with LPCNet was just as good as with Opus, with some caveats. The voices in LPCNet sounded robotic (or "metallic" as one user described it), but LPCNet had less background "fuzz." Opus (at 8 kbit/s) could handle music (whereas LPCNet could not), and sounded like a low quality cellphone connection. Overall, given these two choices, we would deploy Yodel with LPCNet, since Opus does not buy us much at these bitrates.

7 Limitations

Yodel's most significant limitation is its high latency. Perceptual studies [11] show that for conventional interactive telephone-call-like service, 990 ms is likely too high, and would make it difficult to carry on fast conversations with frequent interruptions. However, we believe that Yodel is nonetheless useful for voice conversations with less frequent interruptions, and in our experience, we were able to carry on long conversations over Yodel. We believe that users who value strong call metadata privacy may also tolerate this sort of coarse-grained interactive communication.

Very large deployments. The amount of noise that every Yodel server needs to generate grows linearly with the number of servers, which becomes a bottleneck in deployments with several thousands of servers (e.g., at the scale of Tor, which has 6000 servers). We believe that Yodel is practical despite this limitation. The noise has relatively low overhead in deployments with hundreds of servers (shown in Figure 9 and evidenced in §6.2), which is comparable to the deployments considered in prior work [13, 16].

Fault tolerance. The more hops that a Yodel circuit includes, the more likely it is that a message routes through a server that's down. For example, in §6 we evaluated Yodel with 15 hops. Under this deployment, if 2% of the servers are down then about 30% of the messages will be lost. Thus, Yodel is useful when only a few servers are down, and most messages make it to their destination. Despite this limitation, we believe that handling up to 2% faults can facilitate practical deployments with hundreds of servers and a few servers that are simultaneously unavailable. In terms of availability, Yodel is a step forward over earlier systems [13, 16, 26, 30] where a single faulty server causes the entire system to halt.

Distant users. In our experimental setup with servers located in the US and Europe, users in Australia and South America would experience higher latency (e.g., an additional 100 ms of one-way latency) compared to what we observed from our clients in Boston. A limitation of Yodel's design is that we can't add servers to a new region to reduce the latency for users in that region, without impacting the rest of the users in the system. For example, if we added a server in Australia to our experiments in §6.1, the end-to-end latency would jump from 990 ms to ~3 seconds for all users in the system, since Yodel's latency is approximately the number of hops times the maximum one-way latency between any two servers. We believe this limitation is inevitable when the anonymity set includes all connected users in the system.

Sybil attacks. An attacker may try to create many circuits in order to DoS the system. To mitigate this risk, a deployment of Yodel could rely on existing mechanisms to prevent Sybils (e.g., user subscriptions or proof-of-work). However, fully addressing this problem requires further research.

8 Related work

Tor [7] supports existing VoIP software with acceptable latency but is vulnerable to traffic analysis [20]. The circuits used in Tor resemble the circuits in Yodel, with three crucial differences that enable Yodel to defend against traffic analysis. First, Yodel targets a specific application (voice calls), so its circuits operate in synchronous rounds to eliminate any useful information from traffic patterns (e.g., timing and message sizes). In Tor, an adversary can infer the path of user messages across relays by correlating the times of incoming and outgoing packets. Second, the synchronous design allows Yodel's circuits to be self-healing, which eliminates any leakage from active attacks. In Tor, an adversary that controls a relay could drop a message going through that relay and then see which voice call dropped a packet as a result, which would correlate the sender and receiver of that message. Finally, our analysis shows that Yodel's circuits need to include at least two honest servers to resist attacks from an adversary that fully controls the network. Yodel's circuits traverse more servers (e.g., 15 hops vs. Tor's 3 hops) to meet this requirement with high probability, even when the adversary has also compromised a significant fraction of the servers.

Herd [18] is a metadata-private VoIP system that defends against traffic analysis, but it assumes the user connects via an honest server. This assumption is problematic for two reasons. First, it requires the user to make a tricky choice about which server to trust when joining the system. Second, it gives an attacker a single obvious target for compromising a user's metadata. In Yodel, users don't have to choose which servers to trust, and the cost to compromise any user in the system is much higher: the attacker must compromise a substantial fraction (e.g., well over 20%) of all the servers. Another difference is that Herd provides a weaker notion of privacy, called "zone anonymity", which limits a user's anonymity set to the set of users that connect to Herd via the same server. In Yodel, the adversary cannot learn which pairs of users are communicating, regardless of which servers they connect to.

Loopix [24] hides metadata by relaying messages through several mix servers and randomly delaying messages at each hop. Longer delays improve security by giving messages more time to mix. However, even short delays of 0.5 seconds on average per hop, across 3 hops, results in some messages that experience 3 or more seconds of latency. The high variance latency makes Loopix a poor fit for voice calls.

Riffle [12] uses a similar *hybrid mixnet* design, where a slow setup phase is used to bootstrap a faster communication phase. In the setup phase, Riffle uses a verifiable shuffle (a CPU-intensive cryptographic primitive) to create a permutation on each server, while defending against active attacks. In the communication phase, servers reuse the permutations across many rounds and identify active attacks using authenticated encryption and accusation. In contrast, Yodel's self-healing circuits rely on honest servers to defend against active attacks, which is far more efficient. For example, Riffle supports 10,000 users with sub-second latency, but its approach does not scale as more users join: the latency grows to 10 seconds with 100,000 users. Other systems that use CPU-intensive cryptographic primitives to protect metadata, like Pung [3], XRD [14], and Dissent [32], also suffer from high latency.

Karaoke [16] is a horizontally scalable messaging system that guarantees differential privacy for metadata by routing messages through a mixnet and adding noise. Karaoke and its predecessors [15, 26, 30] focus on scaling to many users, but Karaoke's minimum end-to-end latency (running with no user load) is 6 seconds, which is too high for voice calls. Furthermore, Karaoke's differential privacy guarantee is a poor fit for voice calls because the high rate of messaging would quickly exhaust a user's privacy budget.

The noise circuits in Yodel serve a simpler purpose than the noise messages in prior systems [16, 26, 30]. Prior systems sample a random number of noise messages to obscure the attacker's observations and statistically bound the metadata leakage for a single conversation. Yodel uses noise circuits to ensure that users' messages will mix with messages whose routes are unknown to the adversary; no metadata leaks once messages are mixed.

9 Conclusion

Yodel is a new system for metadata-private voice calls. Yodel achieves the performance required for voice calls by establishing circuits and relying on symmetric cryptography for message processing. The system ensures user privacy in the circuit-based messaging design through two insights, guarded circuit exchange and self-healing circuits. We analyze Yodel's privacy guarantees, implement the system, and evaluate its performance in deployment over the internet with servers located in several countries. With 100 servers, our experiments demonstrate 990 ms one-way message latency for 5 million simulated users. More information and future work will be available at https://vuvuzela.io.

Acknowledgments

Thanks to Robert Morris, Phil Levis, Tej Chajed, the anonymous reviewers, and our shepherd, Brad Karp, for providing feedback that improved this paper. This work was supported by NSF awards CNS-1413920 and CNS-1414119. David Lazar is supported by an SOSP 2019 student scholarship from the National Science Foundation.

References

[1] M. Adorjan. AWS inter-region latency, 2019. URL https://cloudping.co.

[2] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP)*, pages 67–74, Austin, TX, Nov. 1975.

[3] 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.

[4] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *Proceedings of the 2nd International Conference on Cryptology and Information Security in Latin America (LATINCRYPT)*, pages 159–176, Santiago, Chile, Oct. 2012.

[5] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, Sept. 2004.

[6] B. Boudreau. Global bandwidth & IP pricing trends, 2017. URL https://www2.telegeography.com/hubfs/2017/presentations/telegeography-ptc17-pricing.pdf.

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

[9] J. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[10] S. Humphreys and M. de Zwart. Data retention, journalist freedoms and whistleblowers. *Media International Australia*, 165(1):103–116, 2017. URL https://doi.org/10.1177/1329878X17701846.

[11] International Telecommunication Union. G.114: One-way transmission time, Nov. 2009. URL https://www.itu.int/rec/T-REC-G.114.

[12] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the 16th Privacy Enhancing Technologies Symposium*, Darmstadt, Germany, July 2016.

[13] 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.

[14] A. Kwon, D. Lu, and S. Devadas. XRD: Scalable messaging system with cryptographic privacy. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Santa Clara, CA, Feb. 2020.

[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. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 711–726, Carlsbad, CA, Oct. 2018.

[17] D. Lazar, Y. Gilad, and N. Zeldovich. Privacy analysis for Yodel, Aug. 2019. URL https://vuvuzela.io/yodel-analysis.pdf.

[18] S. Le Blond, D. R. Choffnes, W. Caldwell, P. Druschel, and N. Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proceedings of the 2015 ACM SIGCOMM Conference*, pages 639–652, London, United Kingdom, Aug. 2015.

[19] J. Mayer, P. Mutchler, and J. C. Mitchell. Evaluating the privacy properties of telephone metadata. *Proceedings of the National Academy of Sciences (PNAS)*, 113(20):5536–5541, 2016.

[20] 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.

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

[22] C. Nocturnus. Operation soft cell: A worldwide campaign against telecommunications providers, June 2019. URL https://www.cybereason.com/blog/operation-soft-cell-a-worldwide-campaign-against-telecommunications-providers.

[23] Office of the Director of National Intelligence. Statistical transparency report regarding use of national security authorities (calendar year 2018), Apr. 2019. URL https://www.dni.gov/files/CLPT/documents/2019_ASTR_for_CY2018.pdf.

[24] 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.

[25] 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, Philadelphia, PA, Oct. 2016.

[26] 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.

[27] J.-M. Valin and J. Skoglund. A real-time wideband neural vocoder at 1.6 kb/s using LPCNet. arXiv:1903.12087 [eess.AS], Mar. 2019. Available at https://arxiv.org/abs/1903.12087.

[28] J.-M. Valin and K. Vos. Updates to the Opus audio codec. RFC 8251, RFC Editor, Oct. 2017. URL https://tools.ietf.org/html/rfc8251.

[29] J.-M. Valin, K. Vos, and T. Terriberry. Definition of the Opus audio codec. RFC 6716, RFC Editor, Sept. 2012. URL https://tools.ietf.org/html/rfc6716.

[30] 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.

[31] 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.

[32] 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)*, pages 179–192, Hollywood, CA, Oct. 2012.