# Strong and Scalable Metadata Security for Voice Calls

by

David Lazar

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2020

© Massachusetts Institute of Technology 2020. All rights reserved.

Signature redacted

Author ................................
Department of Electrical Engineering and Computer Science
November 12, 2019

Signature redacted

Certified by ...................
Nickolai Zeldovich
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by ...................
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
MAR 13 2020
LIBRARIES
ARCHIVES

# Strong and Scalable Metadata Security for Voice Calls

by

David Lazar

Submitted to the Department of Electrical Engineering and Computer Science on November 12, 2019, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

## Abstract

This dissertation presents a scalable approach to protecting metadata (who is communicating with whom) in a communication system. The emphasis in this dissertation is on hiding metadata for voice calls, but the approach is applicable to any two-way communication between users.

Our approach is embodied in a new system named Yodel, the first system for voice calls that hides metadata 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.

Thesis Supervisor: Nickolai Zeldovich
Title: Professor of Electrical Engineering and Computer Science

# Acknowledgments

I have received support from many people during my PhD. I thank you all in a way that protects our metadata:

`Qc2KNYG0iwErGnLlP82tE1ZLkUe9qR3xkyM0gZUi3wQ`

This hash is the root of a Merkle tree that encapsulates the names of everyone that helped make this dissertation possible. You can verify that you've been acknowledged by obtaining a proof from me, without learning about the other names in the tree. For example, here is a proof that "Alice" is acknowledged:

```
{ Name: "Alice",
  Leaf: 51,
  Salt: "tad5fQhawjDEkGehN4ui9Z-jG9szzp_-bUTYt1AKUEU",
  MerklePath: []string{
    "qkCr_KzgmAJ79W1VaPKbnJlEKuis0LkVPhVwtlswZHI",
    "seMNC_l4LTQaSkQ2OvX7FWea5TkUen9oNsmhyQNfBKk",
    "4Rgoq1k9BjQZt5dQznDrWQ816hDg1CvMfSAmbTXxbmI",
    "u3RPxr_0pk65QyF1qW2s4XC3hecWp4p_ZEwiEzVXPpw",
    "YRZC-1L2WwwzdrYUYsIZv4yI-Izf8VeV0IHQzCoUUNI",
    "1YmY6FFcoRxDQ_D-ZHULdOgYFFsDYvB27iJBHIEknU0",
    "AWvsnoy6sXutTPOU6QUq2cUELLqO0RRil60S241-H9g",},
}
```

Alice can verify the proof with a short piece of code, using the `CheckRecord` function from the tlog package [29]:

`https://play.golang.org/p/rmYTYZtAd24`

To verify that she was thanked in the published version of this document (i.e., on November 12, 2019 and not after the fact), Alice should verify the proof using the root hash that appears in the MIT library's copy of this dissertation.

# Preface

This dissertation is the culmination of my line of work on metadata-private communication:

- ⋆Jelle van den Hooff, ⋆David Lazar, Matei Zaharia, and Nickolai Zeldovich.
  Vuvuzela: Scalable private messaging resistant to traffic analysis.
  In *SOSP 2015.*
- David Lazar and Nickolai Zeldovich.
  Alpenhorn: Bootstrapping secure communication without leaking metadata.
  In *OSDI 2016.*
- David Lazar, Yossi Gilad, and Nickolai Zeldovich.
  Karaoke: Distributed private messaging immune to passive traffic analysis.
  In *OSDI 2018.*
- David Lazar, Yossi Gilad, and Nickolai Zeldovich.
  Yodel: Strong metadata security for voice calls.
  In *SOSP 2019.*

It incorporates ideas that were developed across all of these papers, but is primarily based on our most recent SOSP 2019 paper.

# Contents

# Chapter 1

# Introduction

This chapter motivates the need for metadata privacy, introduces the tension between strong privacy and high performance, and summarizes our key insights and contributions. The next chapter gives a high-level overview of the challenges in protecting metadata and our approach. The remaining chapters go into detail about Yodel, the first system for voice calls that protects metadata from an adversary that controls the network and several of Yodel's servers.

## 1.1 Metadata is highly sensitive

Telecom providers retain call records which include the participants and duration of every call. This metadata is highly sensitive for most users [22] and is especially problematic for journalists who need to keep their sources confidential [13]. As a result, call records are targeted in large-scale attacks [25] and collected by intelligence agencies; the NSA collected 434 million call records of Americans in 2018 [26]. Even if telecoms stop retaining call records, an attacker can monitor the network to learn about voice calls happening in real-time.

## 1.2 Strong security versus low latency

Voice communication requires relatively low latency (a second or two at most) and relatively high bandwidth (a few kilobits per second). However, 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 [5], 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 [21] and Tor [10] 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 [5, 17, 18, 30, 34]. For example, Karaoke [18] 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.

## 1.3 Yodel protects voice call metadata

Yodel is 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, as shown in Figure 1.1. The servers, labeled 1 through $N$, are organized into *layers*, which are indicated by the vertical groups

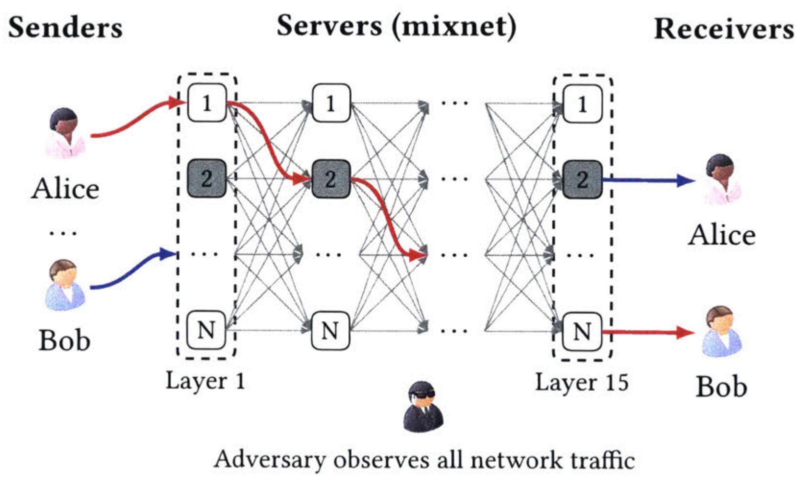


**Figure 1.1:** Yodel's overall architecture consists of users sending messages to servers, which shuffle messages as part of a mixnet, and then send the results back to users. The adversary controls the network and some of the servers (e.g., server 2). Each message takes a random path that is 15 servers long through the mixnet. Layers are an abstraction used to synchronize server-to-server communication as messages traverse the mixnet.

in Figure 1.1. The system operates in *rounds*, and every round all users submit messages to the first layer. Users randomly choose a server on each layer for their messages and use onion-encryption to ensure their messages follow this *path*. At each layer, a server receives onion-encrypted messages from all the servers in the previous layer, removes a layer of encryption from each message, shuffles the messages, and sends the messages to servers on the next layer. The last layer delivers messages directly to users.

Even though all of the user's messages for a round are never processed together on a single server, we show that if there are enough honest servers and enough layers, this design gives the adversary the appearance that all user messages are processed in a synchronous batch. This enables us to hide which pairs of users are communicating in way that is scalable (i.e., the load is distributed among all of Yodel's servers).

Karaoke's design [18] is similar to Figure 1.1, but messages are encrypted with public-key cryptography, which results in significant overheads as messages traverse the network. Yodel amortizes the cost of public-key operations by using symmetric-key circuits through the mixnet to relay many 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.

## 1.4 Key insights

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 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.

## 1.5 Contributions

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) [14], and our prototype uses a low-bitrate vocoder [31], but we find that it provides acceptable voice quality.

The contributions of this dissertation are the following:

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

# Chapter 2

# Overview

In this chapter, we provide a high-level introduction to the challenges in protecting metadata and Yodel's approach to overcoming them. The chapter starts by explaining what we mean by metadata privacy. Then we build up to Yodel's design, starting with a messaging system that runs on a single trusted server, and stepping through the challenges in achieving metadata privacy while also providing good performance. Although Yodel is a system for voice calls, this chapter describes the more general problem of protecting metadata in a messaging system. In Yodel, voice calls are a specialized type of messaging: high-rate with small message sizes.

## 2.1 Metadata privacy

Yodel's security goal is to hide metadata, or who is talking to whom. To clarify this goal, we consider three users who are connected to Yodel: Alice, Bob, and Carol — the remaining users in the system can be under the adversary's control. Figure 2.1 shows the three possibilities for Alice in this case. Either she is talking to Bob; or she is idle and talking to no one; or she is talking to Carol. Yodel's goal is to make these three worlds indistinguishable, so the adversary can't tell which world we are in.

We give the adversary substantial power in trying to figure out which world we are in. We allow the adversary to control the network and, eventually, several

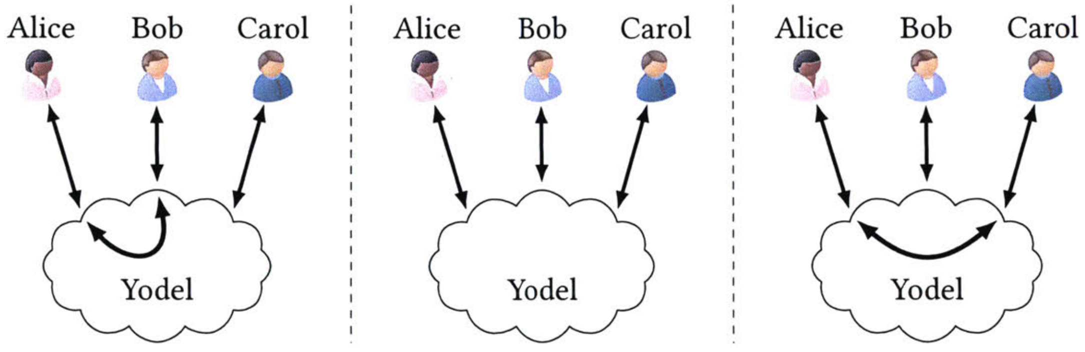


**Figure 2.1:** Yodel's security goal is that an adversary must not be able to distinguish between various possible worlds. In the first world, Alice is communicating through Yodel with Bob. In another, she is connected but not communicating with other users. In a third, she is communicating with Carol.

servers that make up the system. For the remainder of this chapter, the reality is that we are in world 1 where Alice is talking to Bob. To simplify our figures and explanations, we will also assume that Carol is the only other connected user. The goal is to make it equally probable to the adversary that Alice is talking to Bob, to no one, or to Carol, given the adversary's observations and manipulations of the system.

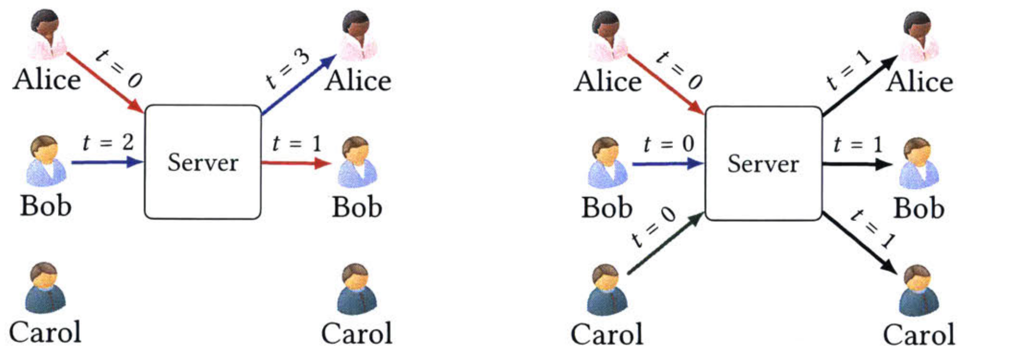
## 2.2 Synchronous cover traffic

Today's most widely used messaging systems are *asynchronous*, meaning users can send messages and receive at any time, as shown in Figure 2.2a. At time $t = 0$, Alice sends a message to the server, which forwards the message to Bob at time $t = 1$. Bob responds to Alice's message a short time later. We assume that Alice and Bob have established a shared key out-of-band, that is used to encrypt and authenticate messages end-to-end. This means the adversary can't distinguish between the messages (arrows) in Figure 2.2a, but it can observe the times at which they are sent.

The adversary can use the timestamps of messages to uncover who Alice is talking to, as part of a *traffic analysis attack*. By monitoring the users' network connections to the server, the adversary sees that after Alice sends a message only Bob receives one, and vice-versa. This leads the adversary to believe that Alice is communicating with Bob. The adversary also concludes that Alice is not communicating with Carol, because Carol doesn't send or receive any messages during this time period even though she is connected to the server. In the asynchronous messaging system, a passive network adversary is trivially able to distinguish between the three worlds from Figure 2.1.

Yodel's solution is to make communication *synchronous* by dividing communication into rounds. Each round, all clients send one message to the server and receive one message back, as shown in Figure 2.2b. Carol is idle, but she sends and receives dummy messages (e.g., to herself) to hide this fact. Similarly, Bob might not have finished typing his message to Alice when the next round starts, so his client will send a dummy message in the meantime. If Bob types a message in the middle of a round, his client queues the message until the next round.

The dummy messages, known as *cover traffic*, are encrypted and indistinguishable from real messages. The synchronous design with cover traffic ensures that network traffic patterns are the always the same, regardless of who users are communicating with, which makes traffic analysis attacks ineffective. Thus, Figure 2.2b meets our security goal against a global passive network adversary because the adversary can't distinguish if Alice is talking to Bob or Carol.



**(a)** Asynchronous messaging **(b)** Synchronous messaging with cover traffic

**Figure 2.2:** An asynchronous messaging system with a single server is vulnerable to traffic analysis attacks, even if the server is honest. Synchronous cover traffic defends against this attack.

## 2.3 Scaling to many users

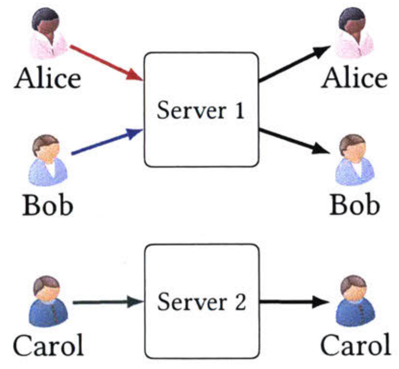


**Figure 2.3:** A messaging system with two servers. The users are sharded among the servers to improve performance, but an adversary can deduce that Alice is not talking to Carol.

As more users join the system, we will need more servers to handle the load, which presents additional challenges. One approach is to "shard" users among multiple servers as shown in Figure 2.3. In this setup, each user connects to one server (e.g., the server closest to them), and messages are routed between servers to their recipients. However, since Carol is idle and sending cover traffic to herself, her messages only pass through server 2. This is problematic because a passive network adversary can deduce that Carol can't possibly be talking to Alice because Carol's server never sends a message to Alice. So this basic approach to distributing the user load fails to meet our security goal.

Ideally, the system should give the adversary the appearance that everyone is communicating in one synchronous batch (as in Figure 2.2b), even though the load is distributed among many servers. Yodel's approach is to route messages through multiple servers, and to have each server shuffle incoming messages before sending them to the next server, which results in multiple plausible routes for each user's message. Figure 2.4 illustrates this approach with two servers and two *layers*. Layers (represented by columns of servers) are an abstraction for synchronizing all server-to-server communication, which protects against traffic analysis attacks. At the start of a round, users pick a random server to process their message at every layer. During the round, every server synchronously collects messages from all the servers on the previous layer, shuffles the incoming messages, and sends the results to servers on the next layer.

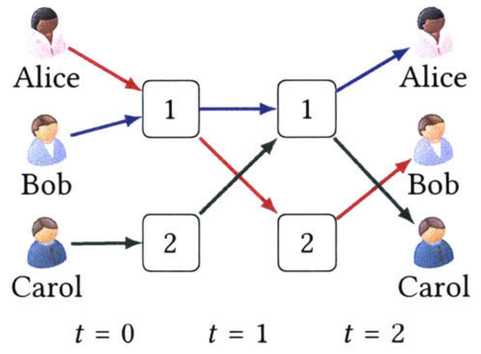
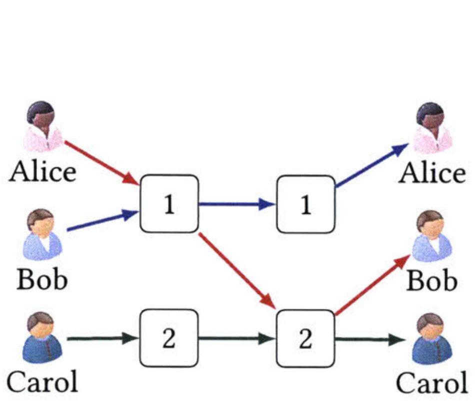


**Figure 2.4:** A messaging system with two servers and two *layers* (vertical columns). Users route their messages through the layers, picking a random server at each layer, with the goal of mixing their messages together. In this case, Alice is plausibly talking to Bob, Carol, or herself (idle).

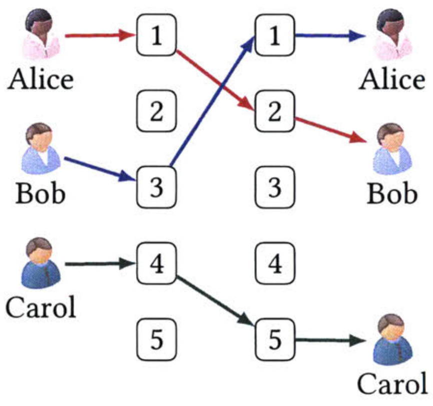
If all clients choose their routes through the layers randomly and independently, then the user load will be distributed evenly among the servers on each layer, enabling Yodel to achieve scalability. For example, in Figure 2.4, Carol's client chose server 2 to handle her message on the first layer, and server 1 on the second layer. The result of this choice is that, even though Alice is actually chatting to Bob, it is equally probable that Alice is talking to Carol based on the network trace. Messages are indistinguishable from the adversary's perspective, so it's possible that Alice sent a message with the path Alice $\longrightarrow$ 1 $\longrightarrow$ 1 $\longrightarrow$ Carol and Carol sent a message with the path Carol $\longrightarrow$ 2 $\longrightarrow$ 1 $\longrightarrow$ Alice and Bob sent a message to himself.

Alice and Carol got lucky that round. What if Carol's message had taken a slightly different path, as in Figure 2.5a? In this scenario, the network adversary can tell that Carol could not have possibly sent a message to Alice, so it is unlikely that they are chatting. This failure to "cross paths" is amplified when there are many servers in the system, as in Figure 2.5b.

Yodel addresses this problem in two ways, which are illustrated in Figure 2.6. First, Yodel servers generate noise messages (represented by black arrows) that



**(a)** 2 servers, 2 layers



**(b)** 5 servers, 2 layers

**Figure 2.5:** Carol's randomly chosen paths do not allow for the possibility that she is communicating with Alice.

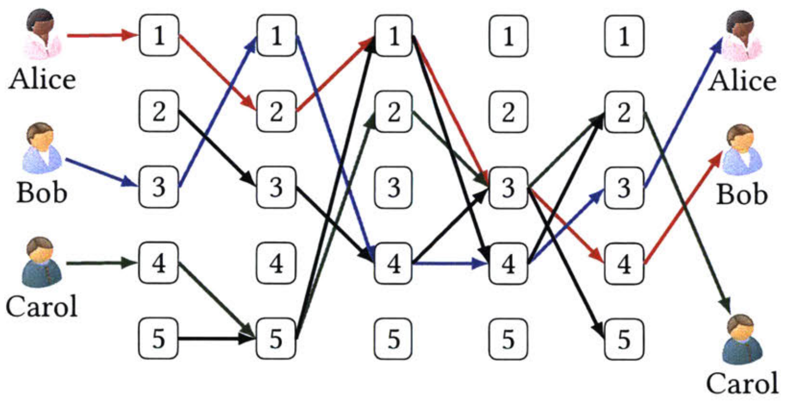


**Figure 2.6:** Yodel uses several layers and random noise messages (black arrows) to ensure that every user has a plausible path to any other user, with high probability.

are indistinguishable from user messages. When a user's message passes through a server where it is shuffled together with another user's message or a noise message, it results in multiple plausible output paths for the message (assuming the adversary doesn't know the order in which messages were shuffled). Yodel's second technique is to use several layers, to give user's messages many opportunities to *mix* with noise messages and each other. To appreciate why these techniques are effective, imagine Figure 2.6 from the perspective of a passive network adversary, where all messages are indistinguishable (i.e., the arrows have no colors). From this perspective, it is impossible to trace an input message on the first layer to an output message on the last layer.

## 2.4 Defending against server compromise

So far we have considered an adversary that can monitor the whole network. Now we consider an adversary that can also compromise several of Yodel's servers. By compromising a server, the adversary learns how the server's input messages correspond to its outputs, so the server provides no security value on a user's path. A compromised server might also try to mis-route users' messages to other compromised servers, or skip generating required noise messages. In general, dealing with malicious servers imposes a significant performance cost on systems that aim to protect metadata. This section gives a high-level overview of Yodel's techniques for dealing with compromised servers and §2.5 describes how we reduce their costs to achieve low latency.

Yodel requires that some of the servers on a user's path are honest to guarantee privacy. If many servers are assumed compromised (and users don't know which ones), then Yodel uses more layers to ensure that paths include honest servers with high probability. Specifically, our approach is to precisely compute the probability that two users' paths become indistinguishable (i.e., get mixed) by the last layer, given the number of layers in the system, the number of noise messages, and the trust assumption. Then, we set the number of layers and noise messages high enough so that any user can plausibly claim they were talking to any other user. For example, we prove (in §5) that if any server is compromised with 20% chance, then deploying Yodel with 15 layers ensures that Alice can claim she was talking to any other honest user with overwhelming probability.

To prevent a server from routing messages maliciously, Yodel puts routing decisions into the hands of users. Each server has a public key known by all users ahead of time. When a user chooses a random path for their message through the layers, they enforce the path by repeatedly encrypting their message using the public key of each server on their path. The resulting *onion encrypted* message, can only be decrypted if it follows the user's path, preventing an adversary from bypassing potentially honest servers along the way. By forcing messages to traverse servers which shuffle messages, Yodel forms a *mixnet*. As we have seen, Yodel's mixnet is unique because it is synchronous to defeat traffic analysis and distributed among many servers for scalability.

## 2.5 Achieving low latency

The Yodel mixnet described so far requires a public key decryption for every message at each of 15 layers. This results in significant end-to-end latency for messages, which makes the system unsuitable for voice calls. Yodel's approach to reducing latency is to split each round into two phases. First, in the *setup phase* users create paths through the layers using public key cryptography. Then in the *messaging phase* users reuse their paths for many messages (e.g., voice packets). Reusing a path requires only fast symmetric key cryptography, which significantly reduces latency but carries risks.

During the setup phase, each server derives a symmetric key for each incoming message that is used to remove a layer of onion encryption. The servers remember the symmetric keys used for each incoming message and the order in which messages are shuffled at each layer. During the messaging phase, servers reuse the symmetric keys to decrypt messages that arrive on the same path, then servers re-apply the saved shuffle so that messages stay on their established paths, and finally servers send their messages to the next layer. Decrypting messages with the symmetric keys is an order of magnitude faster than deriving the key in the first place, which significantly reduces latency during the messaging phase. We refer to these reusable paths as *circuits*.

In Figure 2.7, Alice and Bob have each created one circuit through Yodel's mixnet. Alice and Bob are in a conversation, so they are both connected to each other's circuits on the last layer, which enables them to receive each other's messages. Carol is idle, so she connected to her own circuit to preserve cover traffic. Messages are sent through the circuits synchronously but spend less time at each layer (relative to the public-key encrypted messages used to set up the circuit), since circuits use only fast cryptography.

Figure 2.7 shows the best case scenario, where Alice's and Bob's circuits cross paths at an honest server (server 4 in layer 3), and similar for Alice's and Carol's circuits (server 3 on layer 4). The result is that all three circuits are indistinguishable by the last layer, so the adversary can't tell who is receiving messages from whom, satisfying our security goal from §2.1. In practice, randomly chosen circuits paths are unlikely to intersect at an honest server on the same layer. So Yodel servers generate *noise circuits*, which are not shown in Figure 2.7, to create more plausible routes for circuits that cross paths at an honest server, similar to Figure 2.6.

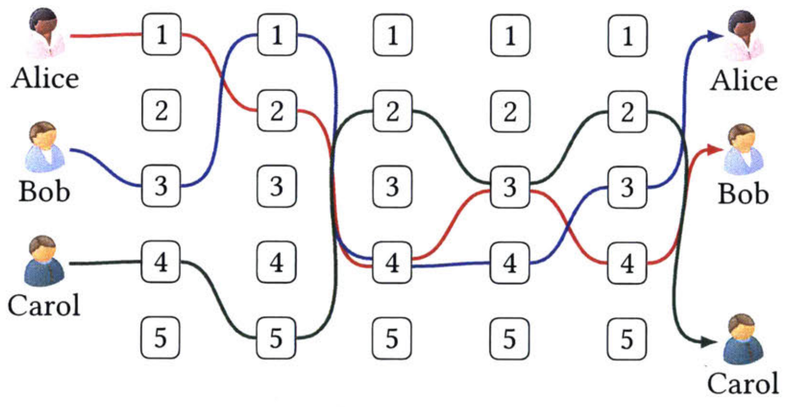


**Figure 2.7:** Circuits are reusable paths through Yodel's mixnet. Alice is connected to Bob's circuit and vice-versa so they can communicate. Two circuits are indistinguishable after they cross an honest server. If servers 3 and 4 are honest, then the adversary can't determine if Alice is talking to Bob, Carol, or herself.

A danger with reusing the same path for many messages is that a user might go offline, which causes an observable gap in the system's network traffic. Suppose that Bob is disconnected, either because he goes offline or because an attacker is blocking his messages. An adversary with a wide view of the network can observe that Alice suddenly stops receiving messages, and thus concludes that Bob was sending messages to Alice.

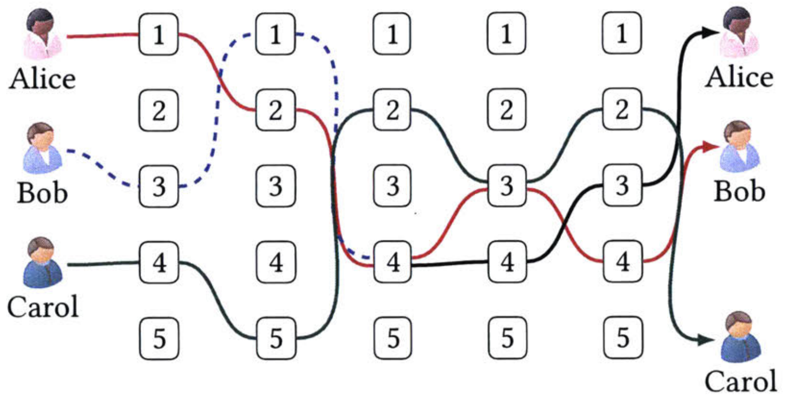


**Figure 2.8:** Bob is disconnected, which creates an observable gap in the network traffic up to the first honest server on circuit. Server 4 is honest, so it fills the gap with an indistinguishable noise message, preventing the adversary from tracing the gap all the way through to its receiver.

Yodel's solution to this attack is *self-healing circuits*. The idea is that an honest server on Bob's path will fill the circuit with noise messages if he ever gets disconnected. For example, in Figure 2.8, Bob was disconnected and server 4 is the first honest server on his path. The server heals the circuit by generating noise messages (black arrows), so the adversary cannot trace the gap in network traffic through to the last layer. The noise messages generated during self-healing are indistinguishable from user messages, so the adversary can't tell which user is receiving noise. Better yet, Yodel guarantees that all circuit output messages from an honest server are indistinguishable, which takes care of other active attacks, including message tagging, duplication, and replay.

This concludes our high-level overview of Yodel's challenges and approach. The following chapters go into detail about Yodel's threat model and design, and how we address additional challenges like:

- How do users learn about their chat partner's circuit?
- What happens when a server goes offline?
- How does Yodel's implementation achieve high throughput over the internet?

# Chapter 3

# 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 [10]; otherwise, the mere use of Yodel may reveal critical metadata [11]. 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.

## 3.1 Security goal

Yodel's security goal is that the probability of an adversary learning any metadata about voice calls is negligible.[1] Specifically, Yodel operates in *rounds* during which each user can start one voice call, and 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

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

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. To limit the information disclosed by the fact that Alice connects to Yodel, we recommend that users run the Yodel client at all times. In principle, users are allowed to connect at any time, but if this correlates with information they are trying to hide, Yodel cannot help. For instance, if Alice and Bob always start their Yodel clients before their daily chat, and then promptly shut down their clients after, an adversary could infer that they are talking. On the other hand, if their Yodel clients are running at all times, an adversary cannot learn when or with whom they are talking. We also aim to support many users so that it isn't suspicious for users to connect in the first place.

## 3.2 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 [31], 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.

## 3.3 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.

## 3.4 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 [24, 28, 35]. For most of this dissertation, 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.

We further assume that honest Yodel clients and servers faithfully implement the Yodel protocol, and that there is no data leakage through side channels. Of course, some clients and servers may be controlled by an adversary (in which case, they need not follow the protocol), but honest clients and servers are assumed to be running bug-free implementations. Yodel's design also assumes that standard cryptographic constructs (e.g., private and public key cryptography and hash functions) are secure.

## 3.5 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 (§7) 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 × 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.

# Chapter 4

# Design

This chapter explains Yodel's design in detail. It starts with an overview of Yodel's components, and then goes into detail using pseudocode for the Yodel client and the Yodel server.

Figure 4.1 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

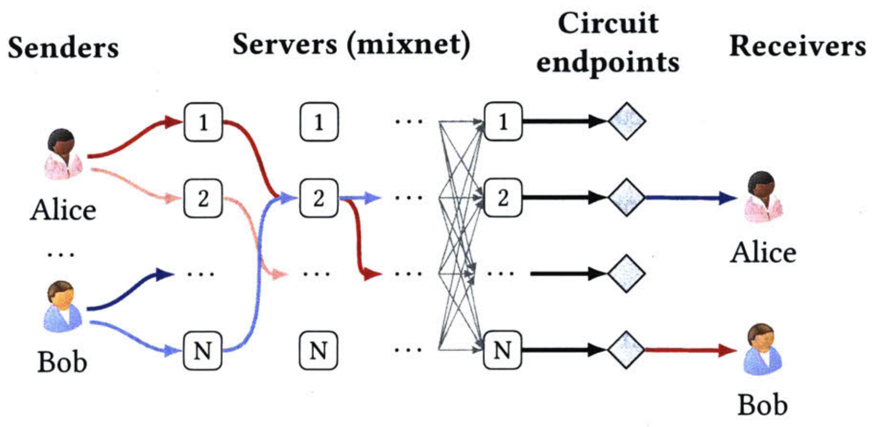


**Figure 4.1:** Overview of Yodel's components. Alice and Bob have created two circuits each. The faded arrows are backup circuits, created as part of Yodel's guarded circuit exchange. Alice and Bob are in a voice call, so they are connecting to each other's circuits, but the adversary doesn't know who created which circuit.

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 4.1, 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 [15, 18]. 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 4.1, 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 [20]). 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 4.1, 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 sub-rounds. 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.

## 4.1 Yodel round pseudocode

New rounds are kicked off by one of Yodel's servers that acts as a coordinator and notifies the other servers about the new round. 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). Clients connect to one of the Yodel servers to receive notifications about new rounds; this server is known as the client's *entry* server. Every time the servers announce a new round, the `client_round` function in Figure 4.2 is called with the new round number, the public keys of the Yodel servers, and the user's call buddy.

Round numbers must increase with every announcement so that the coordinator cannot announce the same round multiple times, but this check is not shown in Figure 4.2. The `onion_keys` parameter is an array with a unique public key for each of Yodel's $N$ servers, freshly generated for the round.[1] 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 4.2. 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.

## 4.2 Circuit setup

During every circuit setup phase (phase 1 in Figure 4.2), 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 4.3 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.

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

  receipts = send_setup_onions([onion1, onion2])
  h1 = onion_decrypt_aes(circuit1.keys, receipts[0])
  h2 = onion_decrypt_aes(circuit2.keys, receipts[1])
  if h1 != hash(circuit1.endpoint) or
     h2 != hash(circuit2.endpoint):
    raise Exception("hash mismatch; aborting round")

  ### Phase 2: Noise Verification
  counts, sigs = recv_noise_signatures()
  noise = 0
  for i in range(servers):
    if verify(sigs[i], servers[i].signing_key):
      noise += counts[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 4.2:** Pseudocode for the Yodel client. Several details (e.g., MACs, nonces, and key rotation) are omitted for clarity.

```
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 4.3:** 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.

The `onion_encrypt` function (Figure 4.3) 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 4.2).

```
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 4.4:** Server pseudocode for circuit setup. The noise generation and verification steps (§4.3) happen before and after this code runs, and are not shown here.

**Server-side processing** Figure 4.4 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.

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 4.4, 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 4.2). 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.

## 4.3 Noise generation

Yodel's privacy guarantee relies on unlinking the user who creates a circuit from the circuit's endpoint. In §5 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 [18]) 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 random paths through the mixnet for their noise circuits and verify that the circuits were established by checking their receipts (as described in §4.2). Our analysis computes the number of noise circuits that every server needs to create to ensure Yodel's security goals (§5).

## 4.4 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[2] 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 number of noise circuits that each server vouched for, as shown in phase 2 of Figure 4.2. The number of noise messages 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 counts 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 §9) 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 $0.6 \times 1.7 > 1$).

[2]An aggregatable signature scheme like BLS [7] could save bandwidth.

## 4.5 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 4.2. In the case that Alice is calling Bob, she sends Bob the endpoint ID of one of her circuits (`circuit1`) through the external messaging 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 4.2), 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 [4, 12]. 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 [5] offer strong security,

```
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 4.5:** Server pseudocode for circuit messaging.

but it could take several minutes to establish a call. Alternatively, Karaoke [18] 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.

## 4.6 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 4.2. 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`.

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 4.5 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 4.6.

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 4.5. 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 4.3), 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 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

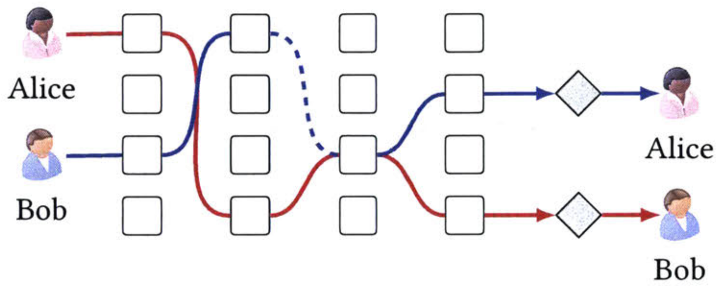


**Figure 4.6:** Self-healing. The dashed line denotes a message that the attacker drops on the blue circuit. The honest server on the third layer fills a message in its place, which ensures that the attacker cannot distinguish red and blue messages after the third layer.

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.

# Chapter 5

# Security Analysis

This chapter formally argues for why Yodel achieves its security goal. Our privacy analysis follows the structure of the client pseudocode from Figure 4.2, 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. §5.1 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 circuits 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.

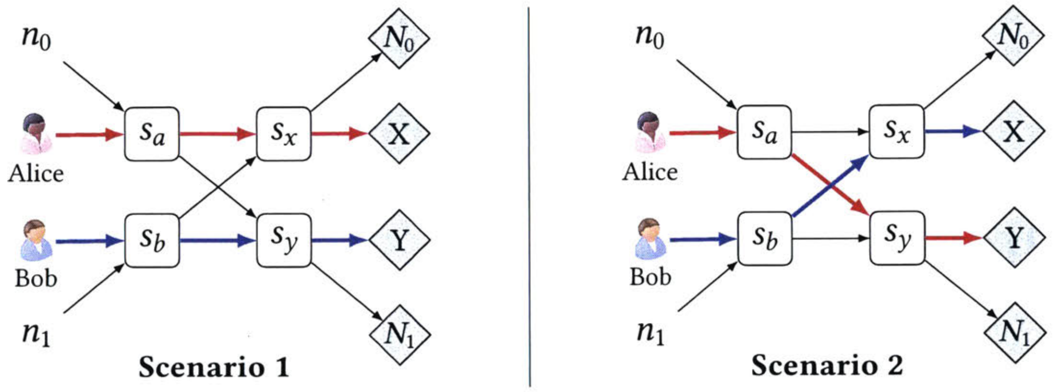


**Figure 5.1:** Yodel's privacy guarantee: an adversary cannot determine whether Alice's message goes to circuit endpoint $X$ and Bob's message to $Y$ or vice-versa (i.e., scenarios 1 and 2 are equally likely given the adversary's observations). Lines represent links with malicious intermediary servers 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 [19] provides a more detailed treatment of Yodel's privacy guarantees.

## 5.1 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 5.1. 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 5.1) is to the left of the rightmost honest server on the other circuit's path (either $s_y$ or $s_x$ in Figure 5.1, 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 §5.2.

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

$$\begin{aligned} &|\Pr[Alice \longrightarrow x \wedge Bob \longrightarrow y \mid \mathcal{O}] \\ &- \Pr[Alice \longrightarrow y \wedge Bob \longrightarrow x \mid \mathcal{O}]| \leq \eta, \end{aligned}$$

*where user $\longrightarrow$ x means that user created the circuit with endpoint x, and $\mathcal{O}$ 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. $\eta$ is a negligible function in the number of noise circuits and the security parameters of Yodel's cryptographic primitives.*

*Proof.* Observe Figure 5.1. 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 5.1. 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 $\frac{1}{N^2}$, where $N$ is the number of servers in Yodel. Thus, 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. $\square$

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 [19] 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 [18, 30, 34]. 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.

## 5.2 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.

The probability that two circuits peer increases with the number of hops in the circuits. Our companion analysis [19] shows how to compute this probability, and Figure 5.2 gives the results for various trust assumptions. The results show that if servers are honest with 80% probability, then two circuits with 15 hops

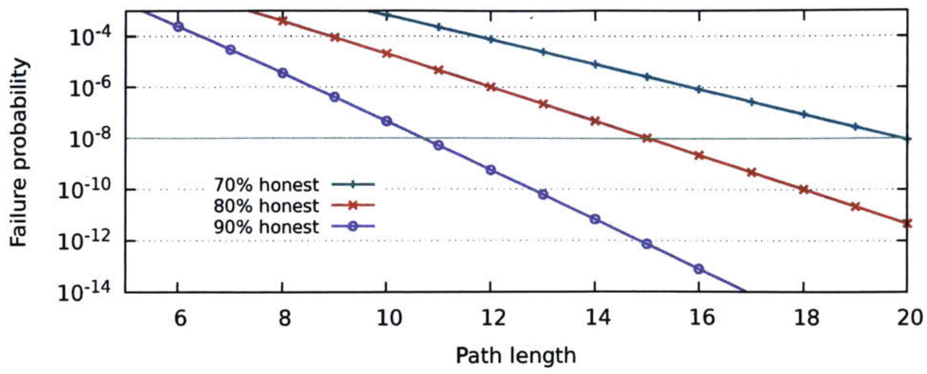


**Figure 5.2:** Probability for two circuits *not* peering as a function of the path length for varying trust assumptions. Our experiments target a failure probability of $10^{-8}$.

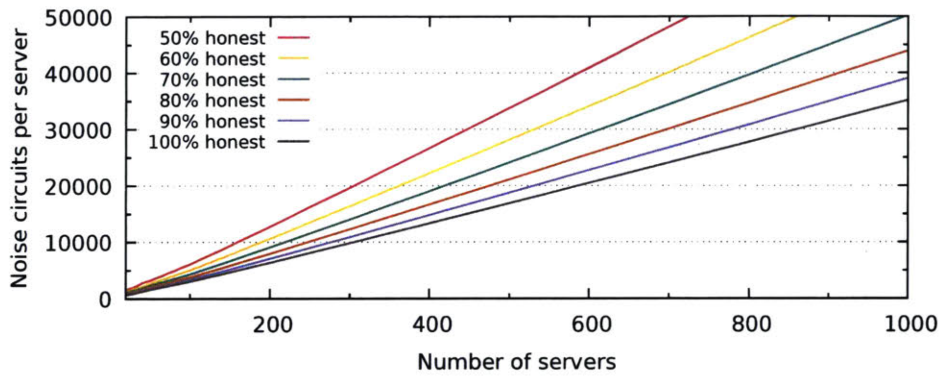


**Figure 5.3:** Number of noise circuits per server needed to guarantee *group privacy* with high probability ($1 - 10^{-8}$), for circuits with up to 20 hops. The number of noise circuits is dependent on the number of servers and the probability of each server being a honest.

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 [15, 18, 27]).

To guarantee group privacy, servers must also create sufficient noise circuits during the circuit setup phase. Figure 5.3 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).

# Chapter 6

# Implementation

To determine if Yodel's design can meet our performance goal (voice calls with sub-second latency for millions of users), we implemented a prototype of Yodel. This chapter describes our Yodel implementation, which is distributed as part of Vuvuzela at https://github.com/vuvuzela.

Our implementation of Yodel is around 10,000 lines of Go code. It uses the NaCl box primitive [6] 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,[1] 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

[1]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.

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 (§7).

Our implementation supports two audio codecs: LPCNet [31] and Opus [32, 33]. 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.

# Chapter 7

# Evaluation

This chapter experimentally evaluates the Yodel implementation. We 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 90 ms (round-trip); the latencies between servers in Europe ranged from 15 ms to 40 ms [3].

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 [20] dialing protocol to agree on a shared secret out-of-band. Lastly, the Karaoke [18] chat system was used for circuit exchange, but it ran on separate servers.

All experiments, except for those in §7.3, 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 $\leq 6\%$.

## 7.1 Yodel achieves sub-second latency

Figure 7.1 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 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,

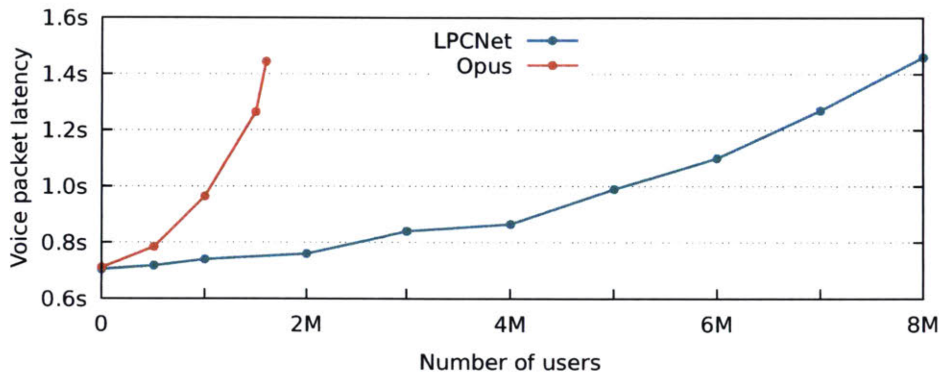


**Figure 7.1:** One-way latency for voice packets with a varying number of users and 100 Yodel servers.

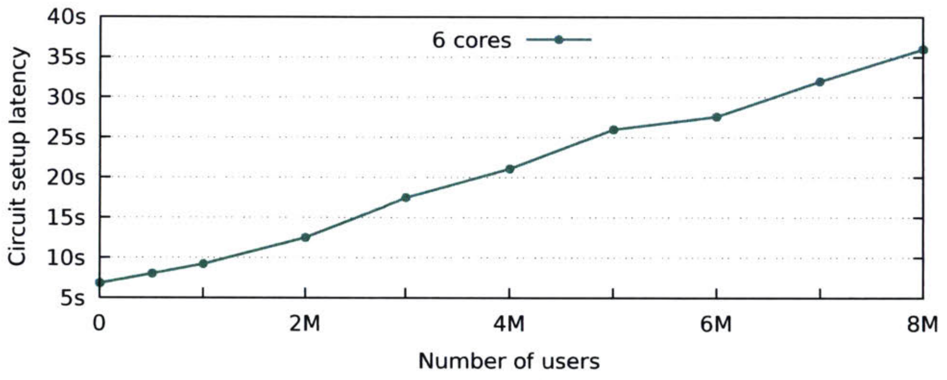


**Figure 7.2:** End-to-end latency of circuit setup with 100 Yodel servers and a varying number of users.

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 LPCNet, hence Yodel is unable to support as many users in this configuration, as shown in Figure 7.1. 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

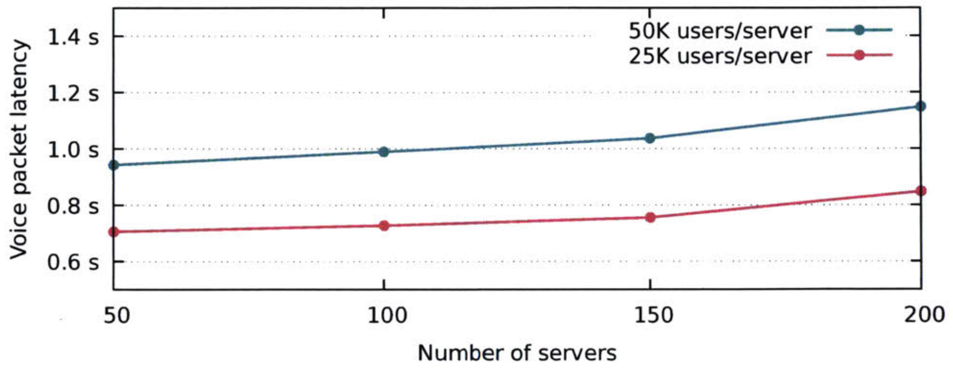


**Figure 7.3:** One-way latency for voice packets with a fixed number of users and a varying number of servers. The right-most points correspond to 5M and 10M users.

setup circuits. Figure 7.2 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.

## 7.2 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 7.3 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 §5). For example, with 200 servers each server is required to create 4× as many noise circuits as in a configuration with

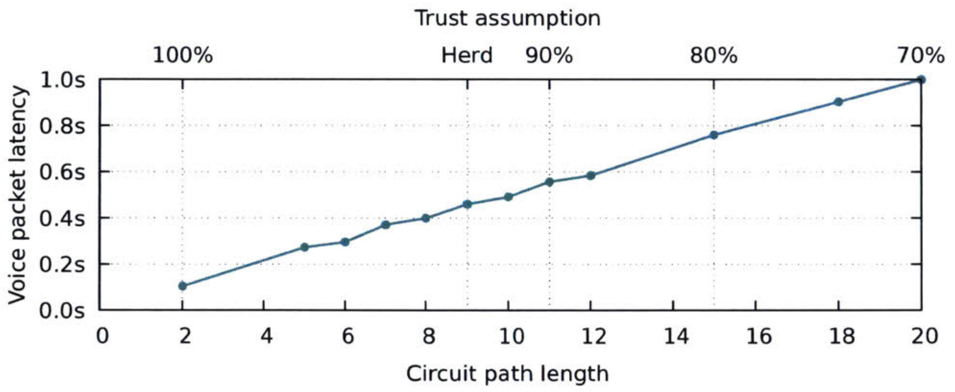


**Figure 7.4:** One-way latency for voice packets with 100 servers, 2 million users, and a varying path length. The top of the graph is annotated with the trust assumptions about server honesty that translate into a given path length.

50 servers, resulting in increased latency. Nevertheless, 200 servers can support 5 million users with 848 ms of latency.

## 7.3 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 5.2. Figure 7.4 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 around 100 ms of latency.

Figure 5.2 also compares Yodel with Herd [21], 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.

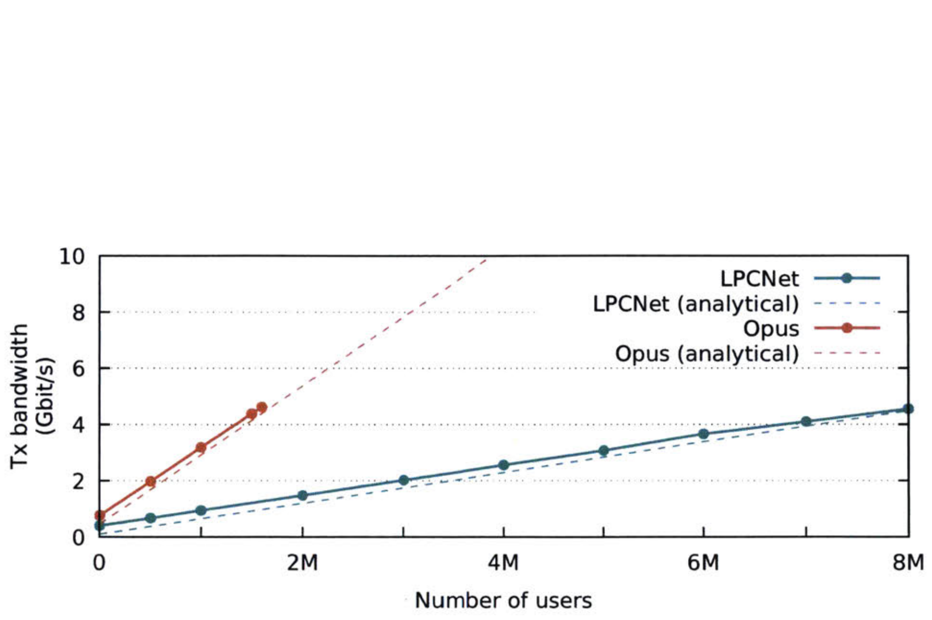


**Figure 7.5:** Average transmit bandwidth per server, with 100 servers and a varying number of users.

## 7.4 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 7.5 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 7.5 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 [8], which suggests that Yodel could charge users less than $1/year to cover bandwidth costs.

## 7.5 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.

# Chapter 8

# Related work

This chapter contrasts Yodel with prior work that aims to protect metadata. We focus on the prior work that comes close to providing strong metadata privacy for voice calls. As we explain in the following paragraphs, the prior work either fails to achieve our security goal or our performance goal. We believe Yodel is the first system to achieve both goals simultaneously.

Tor [10] supports existing VoIP software with acceptable latency but is vulnerable to traffic analysis [23]. 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 [21] 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 [27] 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 [16] and cMix [9] use a similar *hybrid mixnet* design, where a slow setup phase is used to bootstrap a faster communication phase. While Yodel targets a specific application (two-way voice communication), Riffle and cMix are more general but require more complex cryptography as a result. Riffle uses a verifiable shuffle (a CPU-intensive cryptographic primitive) to defend against active attacks during the setup phase, and uses authenticated encryption and accusation to identify active attacks during the communication phase. 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. Neither Riffle nor cMix address the challenge of scaling to many users by adding more servers.

Other systems that use CPU-intensive cryptographic primitives to protect metadata, like Pung [5], XRD [17], and Dissent [36], also suffer from high latency.

Karaoke [18] 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 [20, 30, 34] 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 [18, 30, 34]. 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.

# Chapter 9

# Limitations

This chapter discusses the major limitations of Yodel's design and proposes new directions for future work.

## 9.1 High latency

Perceptual studies [14] 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.

The time to start a new call is also high relative to voice systems which don't hide metadata. In Yodel's experimental setup from §7, users might have to wait up to 5 minutes to start a new call. Yodel's call setup latency could be reduced by running rounds more frequently (e.g., every minute), but we leave it to future work to evaluate how this would impact voice call latency.

## 9.2 Mobile clients

The Yodel client uses bandwidth and CPU in the background to maintain cover traffic, even when the user is not actively talking to anyone. This makes it difficult to run the client on a phone where bandwidth and battery life are limited. Yodel's constant cover traffic (equivalent to one voice call all the time), is ideal for security but quickly drains a phone's battery. Future work should investigate whether Yodel clients could get away with less-than-constant cover traffic without sacrificing usability or security. Recent developments in low-power mobile networking might also be applicable.

## 9.3 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 5.3 and evidenced in §7.2), which is comparable to the deployments considered in prior work [15, 18].

## 9.4 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 §7 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 [15, 18, 30, 34] where a single faulty server causes the entire system to halt.

## 9.5 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 §7.1, 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.

## 9.6 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.

# Chapter 10

# 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`.

# Bibliography

[1] *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[2] *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[3] Matt Adorjan. AWS inter-region latency, 2019.

[4] 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, November 1975.

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

[6] Daniel J. Bernstein, Tanja Lange, and Peter 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, October 2012.

[7] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, September 2004.

[8] Brianna Boudreau. Global bandwidth & IP pricing trends, 2017.

[9] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri De Ruiter, and Alan T. Sherman. cMix: Mixing with minimal real-time asymmetric cryptographic operations. In *Applied Cryptography and Network Security*, pages 557–578. Springer International Publishing, 2017.

[10] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, San Diego, CA, August 2004.

[11] Zach Dorfman. Botched CIA communications system helped blow cover of Chinese agents. *Foreign Policy*, August 2018.

[12] Jim 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.

[13] Sal Humphreys and Melissa de Zwart. Data retention, journalist freedoms and whistleblowers. *Media International Australia*, 165(1):103–116, 2017.

[14] International Telecommunication Union. G.114: One-way transmission time, November 2009.

[15] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* [2], pages 406–422.

[16] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the 16th Privacy Enhancing Technologies Symposium*, Darmstadt, Germany, July 2016.

[17] Albert Kwon, David Lu, and Srinivas 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, February 2020.

[18] David Lazar, Yossi Gilad, and Nickolai 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, October 2018.

[19] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Privacy analysis for Yodel, August 2019.

[20] David Lazar and Nickolai Zeldovich. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *Proceedings of the 12th USENIX*

*Symposium on Operating Systems Design and Implementation (OSDI)* [1], pages 571–586.

[21] Stevens Le Blond, David R. Choffnes, William Caldwell, Peter Druschel, and Nicholas 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, August 2015.

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

[23] Steven J. Murdoch and George 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.

[24] National Security Agency. Tor stinks. The Guardian, October 2013.

[25] Cybereason Nocturnus. Operation soft cell: A worldwide campaign against telecommunications providers, June 2019.

[26] Office of the Director of National Intelligence. Statistical transparency report regarding use of national security authorities (calendar year 2018), April 2019.

[27] Ania M. Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *Proceedings of the 26th USENIX Security Symposium*, pages 1199–1216, Vancouver, Canada, August 2017.

[28] Amirali Sanatinia and Guevara 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, October 2016.

[29] The Go Authors. Package tlog, 2019. `golang.org/x/mod/sumdb/tlog`.

[30] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* [2], pages 423–440.

[31] Jean-Marc Valin and Jan Skoglund. A real-time wideband neural vocoder at 1.6 kb/s using LPCNet. arXiv:1903.12087 [eess.AS], March 2019. Available at https://arxiv.org/abs/1903.12087.

[32] Jean-Marc Valin and K. Vos. Updates to the Opus audio codec. RFC 8251, RFC Editor, October 2017.

[33] Jean-Marc Valin, K. Vos, and T. Terriberry. Definition of the Opus audio codec. RFC 6716, RFC Editor, September 2012.

[34] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai 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, October 2015.

[35] Philipp Winter, Richard Köwer, Martin Mulazzani, Markus Huber, Sebastian Schrittwieser, Stefan Lindskog, and Edgar Weippl. Spoiled onions: Exposing malicious Tor exit relays. In *Proceedings of the 14th Privacy Enhancing Technologies Symposium*, pages 304–331, Amsterdam, Netherlands, July 2014.

[36] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron 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, October 2012.