Fast, Metadata-private Anonymous Broadcast

by

Simon Langowski

B.S., Computer Science and Mathematics, Purdue University (2020)

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
August 26, 2022

Certified by .....
Srinivas Devadas
Edwin Sibley Webster Professor of Electrical Engineering
and Computer Science
Thesis Supervisor

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

Fast, Metadata-private Anonymous Broadcast

by

Simon Langowski

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This thesis presents Trellis: a mix-net based anonymous broadcast system with cryptographic security guarantees. Trellis can be used to anonymously publish documents or communicate with other users, all while assuming full network surveillance. In Trellis, users send messages through a set of servers in successive rounds. The servers mix and post the messages to a public bulletin board, hiding which senders sent which messages.

Trellis hides all network-level metadata, remains robust to changing network conditions, guarantees availability to honest users, and scales with the number of mix servers. Trellis provides three to five orders of magnitude faster performance and better network robustness compared to Atom, the state-of-the-art anonymous broadcast system with a similar threat model.

In achieving these guarantees, Trellis contributes: (1) a simpler theoretical mixing analysis for a routing mix network constructed with a fraction of malicious servers, (2) anonymous routing tokens for verifiable random paths, and (3) lightweight blame protocols built on top of onion routing to identify and eliminate malicious parties.

We implement and evaluate Trellis in a networked deployment. With 32 servers located across four geographic regions, Trellis achieves a throughput of 200 bits per second with 100,000 users. With 64 servers, Trellis achieves a throughput of 320 bits per second. Trellis's throughput is only 100 to 1000× slower compared to Tor (which has 2M daily users) and is therefore potentially deployable at a smaller "enterprise" scale. Our implementation is open-source.

Thesis Supervisor: Srinivas Devadas
Title: Edwin Sibley Webster Professor of Electrical Engineering
and Computer Science

Acknowledgments

I would like to thank my labmate and friend Sacha Servan-Schreiber for taking me under his wing as a new graduate student. We spent many long nights working together and shared lots of failed and successful ideas on the whiteboard. Similarly, to my advisor, Srini Devadas, thanks for brainstoring destinations for many research paths by seeing the valuable nuggets which invite further exploration.

Thank you to all my other labmates: Mayuri, for appreciating my sense of humor; Jun, for knowing that the sausage pizza is the best pizza; Kyle, for great dinners and showing me how to be a good PRIMES mentor; Jules, for giving me interesting assignments; Hanshen, for answering my statistics questions; Albert, for giving us the seed ideas for this project and answering many of my questions; Andrea, Yu (Alex), and Zack, for all sorts of discussions, academic and otherwise. I would also like to thank my friends at MIT, especially Ajay for running board games night, Nikola for teaching me about fully homomorphic encryption, and Derek for always being open to chat about research.

To my friends in Boston and elsewhere: Herrick and Sydney, thanks for continuing our tradition of eating dinners together, after our undergraduate years and into post-pandemic form. Cynthia and Chris, thanks for being a friendly face in a new city and always being willing to get dim sum. David, Brandon, and Kyle, thanks for the good games and keeping in touch. Finally, I would like to thank my family: my brother Eric, for putting up with his younger brother, my parents for their infinite support, and my grandma for writing me letters.

Contents

Chapter 1

Introduction

Communication on the internet is prone to large-scale surveillance and censorship [14, 38, 43, 51, 74, 95]. People trying to hide their communication from powerful (e.g., state-sponsored) adversaries rely on anonymity tools such as Tor [32] or I2P [100] to hide their identities [101, 102, 106]. With Tor, users proxy their traffic through a chain of servers which breaks the link between the identity of the user and the traffic destination. With I2P, users proxy traffic through a chain of peer nodes. While these tools serve an important role, and help obfuscate the identity of the user, they can be insufficient against determined adversaries capable of observing large swaths of traffic (e.g., malicious ISPs). Such adversaries (e.g., ISPs or nation states) can easily identify users and the traffic content through metadata such as packet timing, packet size, and other identifying features—even when all traffic is encrypted [9, 13, 37, 46, 49, 60, 77, 82, 84, 98]. State-of-the-art attacks can deanonymize encrypted Tor traffic with upwards of 90% accuracy by analyzing the encrypted packet traffic [9, 46, 82].

Colloquially, there is an idea of perfect anonymity, where someone is wholly unidentifiable and "could be anyone." In practice, however, anonymity comes in many shades based on relaxations of this ideal. At best, users are only anonymous among those who "use the internet" or those that use the communication system. This set of users, for which a system provides anonymity, is the *anonymity set*. Anonymity systems are embedded into the real world, and this presents additional channels for deanonymization. If we consider the case of making an anonymous forum post, these

users must also use the system before the message is posted, and there must be a message-sized transmission along some path from a computer. Furthermore, it is possible for internet traffic to be intercepted or monitored, and for traffic metadata to reveal information about who is sending what to whom.

Many systems focus on a weaker form of anonymous communication, called point-to-point communication. This is appropriate for anonymous messaging, email, etc., where it is okay for the recipient of a message to knows the identity of the sender. This differs from anonymous broadcast, also known as strong anonymity [19]. Perhaps best known for the application of anonymous forums, it can also facilitate anonymous voting, whistleblowing, anonymous browsing, anonymous publishing, and circumvention of oppressive regimes.

This problem has motivated many systems [2, 19, 20, 35, 36, 62, 64, 78, 110] for anonymous broadcast. Anonymous broadcast consists of anonymously posting a message to a public bulletin board and is the most general form of anonymous communication possible; it can be used for file sharing and to instantiate weaker primitives such as anonymous point-to-point communication [27, 63, 66, 67, 87, 94, 103, 105]. Anonymous broadcast provides *sender* anonymity: fully hiding which user sent which message. Most mix-net based systems, in contrast, only provide *relationship* anonymity [5, 27, 63, 66, 67, 87, 94, 103, 105] (hiding who is communicating with whom). On the other hand, other systems [19, 20, 78, 110] exploit a *non-collusion* assumption (a weaker threat model) to instantiate anonymous broadcast.

As with prior work, Trellis is metadata-private, which guarantees that sender anonymity is preserved in the face of an adversary monitoring the entire network. Trellis also provides horizontal scalability—the ability to increase throughput with the number of (possibly untrusted) servers participating in the network. Horizontal scalability is key to real-world efficiency: Tor and I2P are only capable of handling millions of users [72] on a daily basis thanks to similar scalability guarantees. Moreover, horizontal scalability is important for real-world security: the more users an anonymity system can support, the more "plausible deniability" each user in the system has.

The state-of-the-art system for anonymous broadcast with horizontal scalability is Atom [62]. Unfortunately, Atom suffers from practical barriers to deployment even though it is *theoretically* optimal. (See Section 2.2 for overview of related work.) For example, Atom achieves a throughput of roughly 0.01 bits per second and is six to eight orders of magnitude slower than practical systems such as Tor [72].

All other systems either focus on weaker anonymity notions (e.g., point-to-point communication [63, 66, 67, 94, 103, 105]) or sacrifice scalability by assuming non-colluding servers [2, 19, 20, 35, 36, 61, 78, 110].

1.1 Trellis

Trellis is designed for *fast* anonymous broadcast with *large messages*[1]. Trellis is inspired by a long line of work [16, 62, 64, 90] which augments mix-nets to instantiate anonymous broadcast. Trellis innovates on these works in three important ways: (1) Trellis decouples the use of expensive cryptography required for system setup from the broadcasting phase, (2) Trellis is robust to malicious parties and server churn, and (3) Trellis provides message delivery guarantees even when a *majority*[2] of servers and users are malicious (note that performance degrades with a larger proportion of adversarial servers; see Section 6.2). With these three key contributions, Trellis brings metadata-private anonymous broadcast a step closer to real-world deployment. However, we note that Trellis is still limited by challenges *outside* of the system design itself, such as requiring that all users remain online and contribute messages to ensure metadata privacy. These limitations will be present in any anonymous communication system design with strong anonymity guarantees [43]. Even so, in Section 6.2, we find that Trellis is only 100 to 1000× slower compared to systems like Tor and I2P, both of which sacrifice strong anonymity in favor of performance.

[1] Trellis can be thought of as delivering one large message over many rounds, or providing psuedonymity.

[2] In the case of a dishonest majority of servers, Trellis reconfigures itself into a new network containing a dishonest *minority*; see Section 4.2.2.

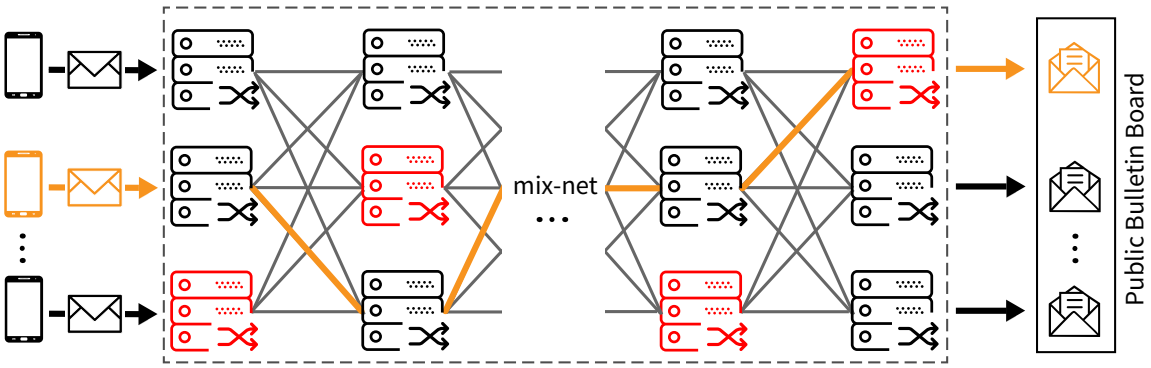

Figure 1-1: Trellis arranges the set of servers into "layers," (each server appears once per layer) to instantiate a mix-net. Users send messages through a random path which guarantees that all messages are mixed before reaching the public bulletin board, even when a subset of servers are malicious (colored in red).

1.2 Ideas, challenges, and contributions

We start by using a mix-net topology known as a random routing network [62, 64]. In such mix-nets, servers are organized into L layers (all servers appear in each layer). The user selects a random path through the servers and sends a message through the path for mixing. Adding more servers increases the throughput, which satisfies the horizontal scalability requirement. While this simple idea seems promising at first (and also proposed in prior work [62, 64]), there are several practical hurdles associated with it:

- Malicious users or servers might deviate from protocol, select non-random paths, or otherwise attack the system. The system must both detect disruption and provably blame the responsible parties. Furthermore, these blame protocols must preserving anonymity for honest users.
- Mix-nets traditionally require that all servers are honest to achieve mixing guarantees. Atom [62] resolves this by emulating honest "servers" using random groups of servers. However, this technique is suboptimal and contributes to slow mixing times in practice (see Section 2.2).
- Real-world networks consisting of disparate servers are not perfect: server churn and elimination can occur frequently. Classic mix-nets are **not** designed to gracefully handle and adapt to changing network conditions on-the-fly.

In Trellis, we overcome the above challenges as follows.

Decoupling path establishment from message mixing. Our first idea—which makes Trellis faster than all prior work—is to entirely *decouple* slow path establishment from fast message routing. This decoupling makes Trellis suitable for high-bandwidth anonymous broadcast and throughput comparable to systems offering much weaker privacy guarantees.

Preventing deviation from protocol. To ensure availability and message delivery, we develop two new tools: anonymous routing tokens (Section 3.4.1) and boomerang encryption (Section 3.4.2). Routing tokens force all users to choose random paths through the mix-net. Boomerang encryption ensures that all servers along a path route messages correctly and enables efficient blaming of malicious servers when required. The combination of these two primitives allows us to port our mix-net to a setting with malicious users and servers actively deviating from protocol *without* compromising on anonymity.

Modeling a mix-net with malicious servers. Kwon [64] develops a novel theoretical mixing analysis for routing mix-nets constructed with some fraction of adversarial servers. In Trellis, we use a routing mix-net topology in conjunction with the analysis of Kwon [64] to avoid the inefficiencies of "emulating" honest servers. Additionally, we derive a more straightforward analysis and bound on the number of mix layers required. Our mixing analysis may be of independent interest to other systems in this area.

Efficient blame, server elimination, and network healing. We design Trellis so that honest servers can efficiently detect failures and deviations from protocol, assign blame, and eliminate responsible parties from the network. Our blame protocols (described in Chapter 4) handle malicious servers and users without ever deanonymizing honest users. Trellis gracefully handles network changes (e.g., offline or eliminated servers) using proactive secret sharing (explained in Chapter 4), which permits on-the-fly state recovery following server churn.

Contributions. In summary, this thesis contributes:

1. the design of Trellis: a novel system for anonymous broadcast providing cryptographic anonymity guarantees for users, network robustness, and concrete efficiency,
2. a simpler theoretical mixing analysis for routing mix-nets instantiated with a subset of mix servers that are malicious and colluding with a network adversary,
3. anonymous routing tokens: a new tool for anonymously enforcing random path selection in mix-nets, with conceivable applications to other networking systems,
4. boomerang encryption: a spin on *onion* encryption that allows for efficient proofs of delivery and blame assignment,
5. and an open-source implementation which we evaluate on a network of up to 128 servers, achieving a throughput of 320 bits per second with 100,000 users and 10 kB messages.

1.3 Limitations

Trellis shares the primary limitation of other metadata-private systems [20, 27, 62–64, 66, 78, 103, 105]: it provides anonymity only among honest online users and therefore requires all users to contribute a message in each round—even if they have nothing to send—so as to hide all network metadata [43]. (Users who stop participating correlate themselves with patterns or content of messages that stop being output [24, 25, 112].) Trellis does not, however, preclude the use of any external solutions to this problem (e.g., Buddies [112]).

Chapter 2

Related work

In general, messages can only be mixed by honest parties. There is no way for an adversary without the involvement of an honest party to mix messages and not know how they are mixed. Systems differ then in two ways: the trust assumptions about who is honest, and how these honest parties mix messages. This has motivated two main solutions: mix-nets, and DC-nets. We start by describing mix-nets and their guarantees in Section 2.1, which follow the anytrust model we use. We then describe related work, including DC-nets, and existing use of mix-nets for anonymous communication in Section 2.2.

2.1 Mix networks

In 1981, Chaum [16] developed the first mix network (mix-net) for the purpose of anonymous *email*. The idea behind Chaum's mix-net is simple. A set of servers are arranged in a sequence. Each server has a public key. Each user recursively encrypts the user's message under the sequence of server public keys. We call this recursive encryption a sequence of **envelopes** (a.k.a. onions [32]), with each envelope encrypted under all subsequent public keys in the sequence. All users send the first envelope to the first server in the sequence. Each server, in sequence, decrypts the envelopes it receives, randomly shuffles the decrypted envelopes, and forwards them to the next server in the sequence. This repeats for N layers. The use of onion encryption prevents

intermediate servers from learning which messages they are routing (a server can only decrypt one layer of encryption). At the end of the sequence, the innermost envelope is decrypted to reveal the plaintext message. The mix-net guarantees that the ordering of the messages is uniformly random and independent from the input, which in turn guarantees unlinkability between each user and their submitted message. As such, classic mix-net architectures can be used for anonymous broadcast. Unfortunately, this architecture suffers from security and practicality challenges:

1. Mix-nets do not inherently guarantee metadata privacy: a network adversary observing all traffic can link messages to users through metadata, such as timing information.
2. Mix-nets are vulnerable to active attacks (e.g., dropped envelopes), which can be exploited by malicious servers to link users to their messages [71, 75, 79, 96, 109].
3. Sequential mix-nets do not provide scalability: adding more servers does **not** improve performance since *all* servers must process *all* envelopes.

Scalable mix-nets. Scalable systems built using mix-net architectures are designed around **parallel mix-nets** [45, 90], where users are assigned to a mix-net instantiated from a subset of servers in the network. Adding more servers thus proportionally increases the total capacity of the network. However, naïvely parallelizing mix-nets does not provide privacy, since a message may end up mixed with only a small subset of other users' messages. To avoid this, parallel mix-nets must model a *random permutation network* [22, 107] (a theoretical framework for modeling mixing in a communication network) to guarantee that all messages are mixed [45, 64, 90]. This is the strategy taken in Atom [62].

2.2 Anonymity systems

In this section, we focus on comparing Trellis to other systems that achieve anonymous broadcast. We pay particular attention to mix-net-based systems as other architectures have incomparable threat models.

Anonymous broadcast using mix-nets. Rackoff and Simon [90] put forth the idea of anonymous broadcast using parallel mix-nets based on permutation networks but do not build or evaluate their approach. Kwon et al. [62] are the first to use their ideas to build a prototype system called Atom. A problem with mix-nets instantiated as permutation networks is that they only guarantee mixing if all servers are honest. The insight in Atom is to use **anytrust groups**—sets of random servers such that at least one server in each set is honest with high probability—to emulate a trusted network from groups of possibly malicious servers. That is, each "server" in the mix-net is emulated by an anytrust group selected at random. Instantiating the mix-net in this way comes at a high cost in terms of concrete performance. The problem is that Atom requires expensive zero-knowledge proofs to ensure servers in each group behave correctly. These quickly become computational bottlenecks limiting the throughput and preventing scaling to large message sizes (Kwon et al. [62] only evaluate Atom on up to 160 byte messages). Another problem is that emulating honest servers with anytrust groups wastes resources and introduces large communication overheads. These performance barriers make Atom impractical to deploy.

Quark [64] is a followup proposal for anonymous broadcast that appears in Kwon [64]'s PhD thesis but is primarily of theoretical interest. Like Atom, Quark is designed on top of a random routing mix-net. However, Quark avoids emulating honest servers by developing a novel theoretical mixing analysis for a network containing malicious servers. Unfortunately, Quark does not prevent disruption attacks: any malicious user or server can cause the entire protocol to restart by dropping (or not sending) a message. Moreover, we find that Quark may not be secure: the blind Schnorr multi-signatures around which Quark is designed can be forged using parallel signing attacks [33, 80] (these attacks were discovered *after* Quark). In Trellis, we salvage the theoretical underpinning of Quark and use a routing mix-net as a starting foundation. We also contribute an improved mixing analysis for this routing mix-net when instantiated with malicious servers, inspired by the ideas of Kwon [64] (see Section 3.9 and Section 3.9). Apart from the mix-net, all other components of Trellis are different.

Anonymous communication using mix-nets. Chaum [16], cMix [15], MCMix [4], Karaoke [66], Stadium [103], Vuvuzela [105], XRD [63], are all anonymous point-to-point communication systems built on top of mix-net architectures. Of these, Karaoke, Stadium, and XRD provide horizontal scalability. However, only XRD provides *cryptographic privacy*. Karaoke, Stadium, and Vuvuzela provide *differential privacy* which introduces a host of practical challenges (see discussion of Kwon et al. [63, Section 1]). Other systems, such as Loopix [87], Streams [27], and Tor [32] are examples of *free-route* mix-nets [26, 40, 69, 76, 91], i.e., parallel mix-nets where users asynchronously route messages via a small subset of mix servers. Free-route mix-nets do not achieve metadata privacy in the face of network adversaries and are susceptible to traffic analysis and active attacks [9, 46, 82].

Other architectures. Other anonymous broadcast systems [2, 4, 7, 19, 20, 35, 73, 78, 111] are instantiated under different threat models. Dining cryptographer networks (DC-nets) and multi-party computation [2, 4, 35, 73] require two or more *non-colluding* servers to process messages and prevent malicious users from sending malformed content. Because of this, these techniques inherently do not provide horizontal scalability and impose high overheads on the servers. Other anonymous communication systems [3, 5, 17, 41, 61, 67, 68, 70, 94] focus on specific applications (e.g., group messaging [17] and voice-communication [3, 67, 68]) or focus on one-sided anonymity (e.g., only recipient anonymity [5]). We refer the reader to the excellent survey of systems in Piotrowska's PhD thesis [86].

2.3 Discussion

In an anonymity system, we would like the adversary to "work for us," meaning that they are contributing to the mixing. For example, they could add randomness or computational resources, that should make the system faster overall. If we were to keep adding adversarial servers to a system, is it possible that the system could become more concretely efficient? Unfortunately, the answer is no, because honest servers must somehow verify the actions taken by the adversarial servers. There are

really only two ways to mix messages: (1) within an honest server, (2) between honest servers, where the communication is disguised (e.g., with dummies). This is because the adversary can view the "data path" of the messages it handles and simply track how the messages were mixed. Mixing within an honest server is the traditional methodology, starting with the mix-net design of Chaum [16]. However, this design inevitably encounters problems with scalability.

The alternative method of mixing is to mix messages *between* honest servers. This can be viewed as grouping of all of the honest servers into one entity, and all of the adversary controlled servers into another. Each group passes messages within itself, and also sends messages to the other group. However, in our case, we assume the adversary can view network metadata, and therefore the adversary can see the messages sent within the honest entity. The contents of messages can be easily hidden with encryption, e.g., AES, but the length and pattern of sent messages can only be hidden through dummy messages, introducing a communication overhead. The key insight is that since this honest group is composed of many servers, it can mix messages much more efficiently (and with less overhead) than a single server can. This was explored in Quark [64], but the design lacked robustness. In particular, this method only works if all users are honest, but the contributions of *colluding malicious users* cause Quark to fail. Specifically, any purposely malformed message would require a total system abort and restart. The overarching problem is that the adversary knows the communication paths of the messages of any colluding users. Removing these known contributions can leave the resulting pattern of communication between honest servers (that was supposed to be disguised by dummies) non-uniform.

However, users, are still useful. They are an efficient source of trust: each user is a source of trust for their own message. We will take advantage of this fact when we design boomerang encryption (Section 3.4.2). Since users can be malicious, and we should only assume two are honest, then any user-trust system would have to have all pairs of messages intersect, which would result in an $O(M^2)$ protocol, in both communication and computation. This is in fact exactly what a DC-net does, where all M users contribute to each of M messages. This is still not enough, and DC-

nets require additional server work to verify users do not tamper with each others' messages, due to the way "mixing" works [78].

A more efficient mixing, is to mix not in pairs, but in groups that are as large as possible. In particular, we will choose the largest group, the set of all messages in honest servers at some point in time, as in Quark [64]. We will need to combine server-based trust and user-based trust in order to create a protocol for a dishonest majority, and we will need to tolerate a finite number of restarts. These details are covered in Section 4.2.2.

Once one decides on a mixing method, systems differ in how to verify the actions of servers, and verifying the mixing (or more accurately, proving we can blame for an incorrect mixing), composes the majority of this thesis. Some systems [103] [62] simply use a zero-knowledge verifiable shuffle, but this is slow—quadratic in the number of messages being mixed. Systems with a different threat model, the non-collusion assumption, we can directly assign servers to watch over each other. In Spectrum [78], two servers are assumed to not collude and check each other's work if necessary. Clarion [35] uses a multiparty computation (MPC) based approach where a group of three servers check each other's work when shuffling messages. In our setting, where we intend to have hundreds (even thousands) of servers, the non-collusion model does not make much sense, as an adversary could easily control two servers. Rather, non-collusion models focus on a small number of servers who have public identities, like corporations or universities, that can be reasoned about. Because of our model and desire for performance, we develop much more efficient and lightweight protocols, in Chapter 4. In our case, we must design protocols that tolerate a dishonest majority, eliminating many previous approaches which simply deadlock.

Chapter 3

Trellis

3.1 Notation

General. We denote assignment of a variable b to a as $a := b$. The output of (possible randomized) algorithm Alg is denoted as $a \leftarrow \mathsf{Alg}$ and a random sampling from a distribution $\mathcal{D}$ is denoted as $a \xleftarrow{R} \mathcal{D}$. Vectors are denoted in bold as $\boldsymbol{a}$ where the ith component of the vector is indexed as $\boldsymbol{a}[i]$. We use λ to denote a computational security parameter.

Trellis specific. We denote by N the number of servers in the network and by $\mathbb{S}$ the set of N server identifiers (without loss of generality, we can set $\mathbb{S} = \{1, \ldots, N\}$). The fraction of adversarial servers is f (see Section 3.3 for details). The total number of messages is denoted M and we assume it coincides with the number of users. The total number of layers is denoted by L and we will typically use ℓ to denote the variables in the ℓth layer; e.g., the ℓth server on a given path is denoted S_ℓ.

3.2 Main ideas

As with all mix-net based systems [15, 27, 62, 64, 66, 67, 90, 94, 103, 105], servers are organized into a network of L layers and users' messages are forwarded by servers in each layer (see Figure 1-1). Trellis is organized in two stages. A one-time path

establishment stage followed by a repeated message broadcasting stage. In both stages, Trellis resolves failures and attacks by malicious users and servers through on demand blame protocols (fully described in Chapter 4).

Do once: Path establishment. We reference the steps in Figure 3-1. Path establishment proceeds layer-by-layer. ① Users obtain signed anonymous routing tokens (Section 3.4.1) from an anytrust group of signers which guarantees the resulting path assigned to the user is uniformly random. Each token issued for a layer determines the server at the next layer and contain all the necessary routing information needed to route message between the two layers (details in Section 3.4.1). ② Users distribute the L tokens anonymously using boomerang encryption (Section 3.4.2). All servers obtain the necessary routing tokens without learning the full path.

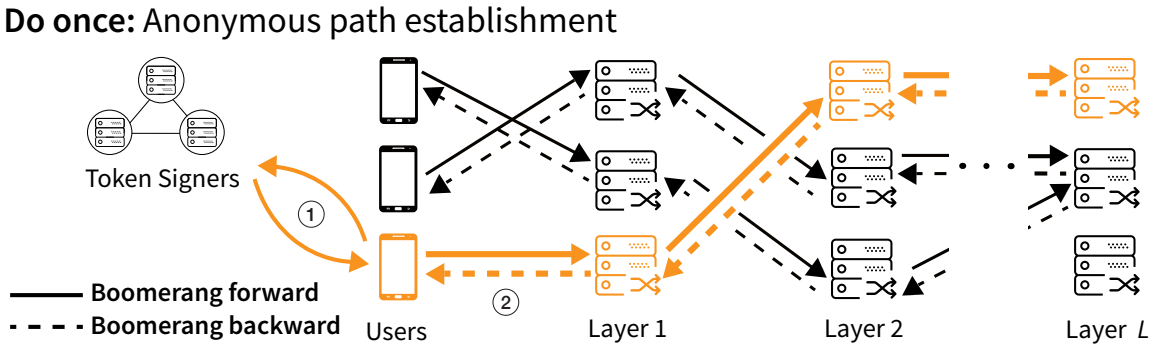

Figure 3-1: Users are assigned a server in each layer through a one-time setup protocol involving an anytrust group of signers.

Repeat: Anonymous broadcasting. We reference the steps in Figure 3-2. Anonymous broadcasting repeats indefinitely on the pre-established path. ① Each user sends their envelope (containing the user's message) through the server assigned by the routing token, for each of the L layers. ② Servers collect all the envelopes, decrypt one layer of encryption, and group the resulting envelopes based on their destination server in the next layer. Servers shuffle and pad each group with dummy envelopes as necessary. Each group is then sent to the destination server in the next layer. ③ Once envelopes reach the final layer, they are guaranteed to be shuffled and unlinakble to the senders, at which point the final layer of encryption is removed to reveal the plaintext message.

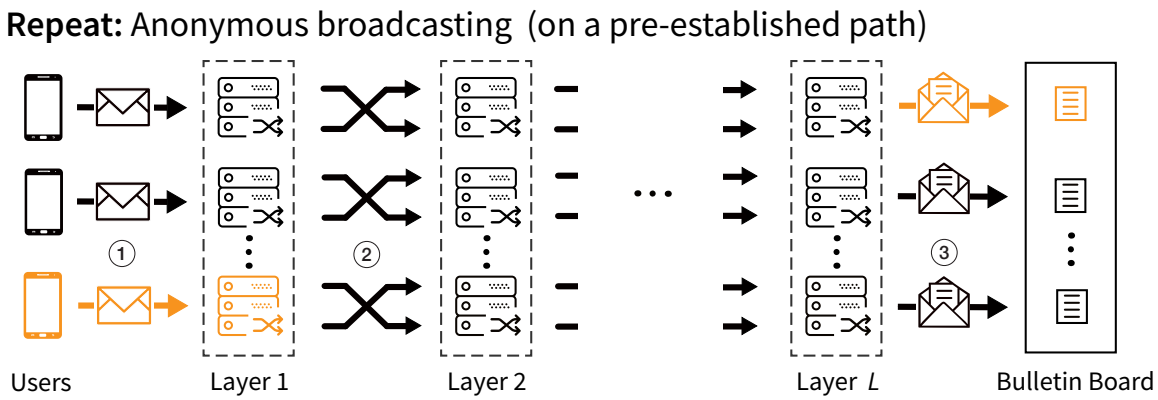

Figure 3-2: Users send messages through the layers, based their paths. Users can repeat this process indefinitely to broadcast many different messages.

On-demand: Blame, elimination, and recovery. It could be the case that a server goes offline or acts maliciously, both during path establishment and the broadcasting stages. To deal with such failures (intentional and otherwise) we develop on-demand blame and recovery protocols that automatically reassign affected paths. Importantly, failure resolution is handled *locally* on the link between two layers and does not involve the users. (Failures caused by a malicious user cause the user to be eliminated by other means; see Chapter 4.) We follow the steps of Figure 3-3. Blame and recovery is invoked whenever an error is detected (e.g., wrong signature, dropped envelopes). ① A server detects an error in the received batch of envelopes in layer $(i{+}2)$ and proceeds to blame the server in layer $(i{+}1)$ that sent the batch. ② Servers evaluate evidence from both parties to decide which of the two is malicious and ③ vote for the honest party by providing a secret share of a the malicious party's secret state. The state is used by replacement server to take over the duties of the eliminated server.

3.3 Threat model and assumptions

The threat model and assumptions underpinning Trellis mirror those of existing anonymous broadcast [62, 64] (and communication [4, 15, 63, 66, 67, 103, 105]) systems based on mix-nets that provide metadata-privacy guarantees.

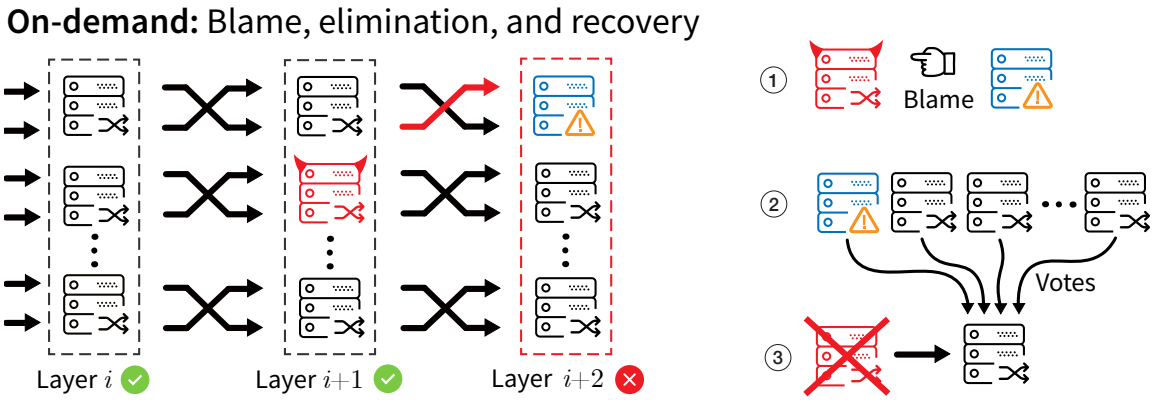

Figure 3-3: On-demand blame and recovery protocols are invoked by servers to automatically reassign affected paths and eliminate malicious servers from the network.

Assumptions and adversarial model.

Observed network. We assume that all communication on the network is observed by a passive network adversary but that the communication *content* is encrypted using standard authenticated public key infrastructure (e.g., TLS [93]).

Malicious users and servers. We assume that a fixed fraction $0 \leq f < 1$ of servers and any number of users are malicious, may arbitrarily deviate from the protocol, and collude with the network adversary. Note: in the case of a dishonest majority ($f \geq 0.5$), our blame protocols in Chapter 4 cause successive network reconfirgurations which converges to an *honest majority* (details in Section 4.2.2).

Cryptography. We make use of standard cryptographic assumptions. We require digital signatures, symmetric-key encryption, and the computational Diffie-Hellman (CDH) assumption [31] (and variants of the CDH assumption for *gap* groups [58]; see Appendix 5.3.2). We also require pairings on elliptic curves [12, 53] and hash functions modeled as random oracles when constructing our routing tokens (Section 3.4.1).

Liveness. We require two liveness assumptions: synchronous communication and user liveness. Both assumptions are standard in all prior work on metadata-private anonymous communication [2, 4, 5, 19, 20, 35, 62–64, 78, 103, 105, 110].

- *Synchronous communication:* Servers are expected to be able to receive all envelopes for each layer before processing the next layer. This is equivalent to assuming all servers have access to a global clock [97].

- *User liveness:* We assume that all users submit a message for every round—even if they have nothing to send—to prevent the risk deanonymization via intersection attacks [24, 25]. We do **not** assume server liveness.

Non-goals. Trellis does not aim to prevent denial-of-service (DoS) attacks on the network or handle infrastructure failures. However, Trellis is fully compatible with existing solutions [18, 21] for network availability and reliability.

Trellis guarantees. Under the threat model and assumptions of Section 3.3, Trellis provides the following guarantees.

Robustness and availability. No subset of malicious servers or colluding users can disrupt availability for honest users.

Message delivery. All messages submitted by honest users are guaranteed to be delivered and result in a "receipt" that can be efficiently verified by each user.

Unlinkability. Fix any small $\epsilon > 0$ and any subset of honest users. After mixing, no message (or set of messages) can be linked with any honest user (or set of honest users) with probability ϵ over random guessing by the adversary.

Metadata privacy. The unlinkability guarantee holds even when all network communication is visible to the adversary who is controlling the malicious users and servers.

3.4 Cryptographic foundations

We briefly describe classic cryptographic primitives that we use as building blocks in Trellis. We point to excellent textbooks for more details on these standard primitives [11, 44, 56].

Diffie-Hellman key agreement. The Diffie-Hellman key agreement protocol [31] allows two parties to establish a shared secret key over an authenticated communication channel. Let $\mathbb{G}$ be a suitable group of prime order p with generator g.

$\mathsf{DHKeyGen}(\mathbb{G}, g) \to (A, a)$:

1: Sample $a \in \mathbb{Z}_p$.

2: Output $(A := g^a, a)$.

$\mathsf{DHKeyAgree}(A, b) \to \mathsf{sk}$:

1: Output $\mathsf{sk} := (A)^b$.

Observe that for all (A, a) and (B, b) sampled according to $\mathsf{DHKeyGen}(\mathbb{G}, g)$, $\mathsf{DHKeyAgree}(A, b) = \mathsf{DHKeyAgree}(B, a)$. Thus, two parties derive the same secret key sk.

Digital signatures. A digital signature (DS) scheme consists of three algorithms $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ satisfying *correctness* and *unforgeability*. Informally, correctness requires Verify to output yes on all valid signatures and unforgeability states that no efficient adversary should be able to produce a forged signature under vk that is accepted by Verify.

Symmetric key encryption. An encryption scheme (e.g., AES [23, 81]) consists of three algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ satisfying correctness and IND-CCA security. Informally, correctness requires that $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Enc}_{\mathsf{sk}}(m)) = m$ for all messages m and secret keys sk sampled according to KeyGen. IND-CCA security requires the distribution of ciphertexts induced by any pair of messages to be computationally indistinguishable *and* authenticated (see Boneh and Shoup [11] for definitions).

Anytrust groups. Let κ be a statistical security parameter. Given a set of servers where some fraction $0 \leq f < 1$ are malicious, there exists an integer s such that the probability that any group of s random servers contains *at least* one honest server is $1 - 2^{-\kappa}$ (proof: it is enough to pick s such that $f^s < 2^{-\kappa}$) [62].

3.4.1 New tool: Anonymous routing tokens

Anonymous routing tokens (ARTs) are used in Trellis to determine and verify routing paths. Our construction of ARTs is inspired by anonymous tokens based on blind signatures [28, 34, 58], but has two key differences: namely ARTs are signed by a *group* of signers and are publicly verifiable. Specifically, tokens are signed by an anytrust group, This guarantees that at least one honest signer is present when signing each token and forms the basis for integrity in Trellis. The blind signature serves two purposes: ensuring that (1) the payload encapsulated by the ART comes from an (anonymous) user, and (2) the path through the mix-net, as determined by the ART, is uniformly random (even for malicious users).

How our routing tokens work. Routing tokens are generated by users. Each token encapsulates the necessary information a server requires to route envelopes on a link in the mix-net. Each signer (a server in an anytrust group) holds a partial signing key x_i associated with a global public signature verification key tvk_ℓ for the ℓth layer. The user has each group member provide a partial signature for the blinded token. The user then unblinds and recovers the multi-signature with respect to tvk_ℓ by combining the partial signatures. If at least one signer is honest, then the signature is unforgeable and cannot be traced to the tokens seen during signing. The randomness of each token signature—which is uniform and independent provided at least one signer is honest—determines how envelopes are routed through the network.

Definition 1 (Anonymous Routing Tokens; ARTs)**.** Fix a security parameter λ, integers $s, N > 1$, a common reference string crs generated using a trusted Setup process, and a set $\mathbb{S}$ of N server identifiers. An ART scheme consists of algorithms KeyGen, $\mathsf{NextServer}$, Verify, and an interactive protocol Sign instantiated between a user and s signers.

- $\mathsf{KeyGen}(\mathsf{crs}) \to (\mathsf{tvk}, x_1, \ldots, x_s)$. Takes as input the crs. Outputs a public key tvk and s partial signing keys.
- $\mathsf{Sign}\langle \mathsf{User}(\mathsf{tvk}, \mathsf{payload}), \mathsf{Signers}(x_1, \ldots, x_s)\rangle \to \langle \mathsf{t}, \perp\rangle$. The user inputs a payload consisting of the server identity $S \in \mathbb{S}$, a **new** Diffie-Hellman message and a **new** digital signature verifcation key. The ith signer provides as input the partial signing key x_i. The user obtains a (signed) routing token t committing to the $\mathsf{payload}$. The signers obtain no output ($\perp$). See Figure 3-4 for an overview.
- $\mathsf{NextServer}(\mathsf{tvk}_\ell, \mathsf{t}_\ell) \to (S_{\ell+1})$. Takes as input the layer public key tvk_ℓ, the token t_ℓ. Outputs the identity of the next server $S_{\ell+1}$ deterministically chosen based on t_ℓ.
- $\mathsf{Verify}(\mathsf{tvk}_\ell, S_{\ell+1}, \mathsf{payload}_\ell, \mathsf{t}_\ell) \to \mathsf{yes}$ or no. Takes as input the public key for layer ℓ, the payload $\mathsf{payload}_\ell$, identity of the next server $S_{\ell+1}$, and signed token t_ℓ. Outputs yes if ($S_{\ell+1}$ and $\mathsf{payload}_\ell$ are valid under t_ℓ and no otherwise.

The above functionality must satisfy the standard properties of blind digital signa-

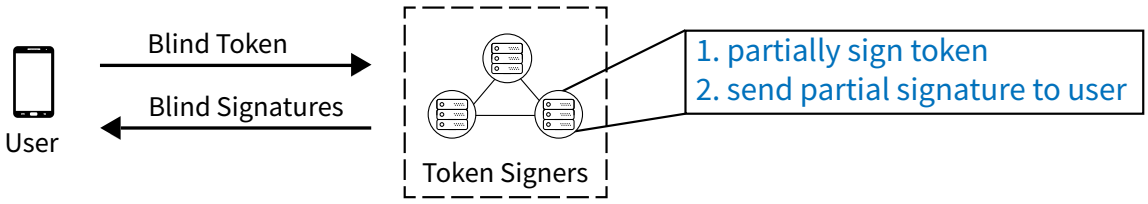

Figure 3-4: Overview of the ART signing protocol (Sign).

tures [88] (and anonymous tokens [28, 58]): *completeness*, *unforgeability*, *unlinkability*, in addition to a new property we call *unpredictability*. Unforgeability ensures that if the ART verifies under the public key tvk, then the encapsulated payload came from some user and was not tampered with. Unlinkability for ARTs guarantees that the encapsulated payload is random and independent of any user (thus the Diffie-Hellman message and a signature verification key are sampled randomly). Finally, unpredictability requires that $S_{\ell+1}$ (as output by $\mathsf{NextServer}$) be unpredictable given the payload. We formalize these properties in Appendix 5.3.1.

ART construction

We realize an ART scheme satisfying Definition 1 following the blueprint of BLS signatures [12] and anonymous tokens [28, 58]. In Appendix 5.3.2, we prove security of our construction under a variant of the computational Diffie-Hellman (CDH) assumption in gap groups [10, 12, 58]. Let $\mathbb{G}$ and $\mathbb{G}_T$ be groups of prime order p with generator g and equipped with an efficiently computable non-degenerate bilinear pairing $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ [11]. We let $\mathsf{crs} := (\mathbb{Z}_p, \mathbb{G}, g) \leftarrow \mathsf{Setup}(1^\lambda)$ for a security parameter λ.

$\mathsf{KeyGen}(\mathsf{crs})$:

1: $x_1, \ldots, x_s \xleftarrow{R} \mathbb{Z}_p$

2: $x \leftarrow \sum_{i=1}^{s} x_i$

3: $\mathsf{tvk} \leftarrow (g^{x_1}, \ldots, g^{x_s}, g^x)$

4: **return** $(\mathsf{tvk}, x_1, \ldots, x_s)$

$\mathsf{NextServer}(\mathsf{tvk}_\ell, \mathsf{t}_\ell)$:

1: $S_{\ell+1} := \mathsf{Hash}_{\mathbb{Z}_N}(\mathsf{tvk}_\ell || \mathsf{t}_\ell)$

2: **return** $S_{\ell+1}$

$\mathsf{Verify}(\mathsf{tvk}_\ell, S_{\ell+1}, \mathsf{payload}_\ell, \mathsf{t}_\ell)$:

1: parse $\mathsf{tvk}_\ell = (g^{x_1}, \ldots, g^{x_s}, g^x)$

2: $y \leftarrow \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload}_\ell)$

3: $S'_{\ell+1} \leftarrow \mathsf{NextServer}(\mathsf{tvk}_\ell, \mathsf{t}_\ell)$

4: **return** $e(\mathsf{t}_\ell, g) = e(y, g^x)$

and $S'_{\ell+1} = S_{\ell+1}$

Token signing protocol. We instantiate the token signing protocol in one round between the user and each signer. The user begins by generating a fresh Diffie-Hellman key exchange message that will later be used to derive the encryption key for its ℓth link. The user also generates a fresh digital signature key pair, which will be used to verify integrity of its encryptions on the ℓth link.

Step 1 (user):

1: $(A_\ell, a_\ell) \leftarrow \mathsf{DHKeyGen}(\mathbb{G}, g)$; $(\mathsf{vk}_\ell, \mathsf{sk}_\ell) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$.

2: $r \xleftarrow{R} \mathbb{Z}_p$; $T_\ell \leftarrow \mathsf{Hash}_\mathbb{G}(\underbrace{S_\ell || A_\ell || \mathsf{vk}_\ell}_{\mathsf{payload}_\ell})^{1/r}$.

3: **send** T_ℓ to all signers.

Step 2 (ith signer): Upon receiving T_ℓ, the ith signer partially signs the blind hash using their share x_i of the signing key:

1: $W_{\ell,i} \leftarrow T_\ell^{x_i}$.

2: **send** $W_{\ell,i}$ to the user.

Step 3 (user): The user combines the partial signatures and outputs an unblinded token:

1: $\mathsf{t} \leftarrow (\prod_{i=1}^{s} W_i)^r$.

The user then checks that t_ℓ was signed correctly:

2: $S_{\ell+1} \leftarrow \mathsf{NextServer}(\mathsf{tvk}_\ell, \mathsf{t}_\ell)$.

3: $\mathsf{Verify}(\mathsf{tvk}_\ell, S_{\ell+1}, \mathsf{payload}_\ell, \mathsf{t}_\ell) \stackrel{?}{=} \mathsf{yes}$.

If Verify outputs no, then the user discards the token and outputs $\perp$. Otherwise, the user stores $(\mathsf{payload}_\ell, \mathsf{t}_\ell, \mathsf{sk}_\ell, a_\ell)$.

Generalization. We remark that our construction of ARTs can be generalized to support more generic anonymous tokens such as the ones described in Kreuter et al. [58] and Davidson et al. [28] but instantiated with a *set* of signers rather than just one signer as in previous work [28, 58, 104]. Additionally, unlike prior constructions, ARTs are *publicly* verifiable. These features make ARTs potentially of independent interest. For simplicity, we restrict our definition of ARTs to Trellis.

3.4.2 New tool: Boomerang encryption

While ARTs verify integrity of the user's routing data, they can't ensure that the ART (or the user's envelopes) gets to the appropriate server on the path. A user complaining of protocol deviation runs the risk of deanonymization. Boomerang encryption—a simple extension of onion encryption [16]—solves this problem by providing receipts that can be used to *anonymously* blame on a user's behalf. Boomerang encryption is designed to (1) ensure that all parties along a path receive messages addressed to them and (2) allow honest servers to trace the source of active attacks and assign blame.

Existing tool: onion encryption and routing. Onion encryption [16, 32] involves recursively encrypting a message under a sequence of keys. Only the corresponding secret-key holders can recover the message by "peeling off" the layer of encryption one by one. Formally, given any symmetric encryption scheme with algorithm Enc, onion encryption is defined as: Formally, given any symmetric-key encryption scheme with algorithm Enc, onion encryption is defined as:

$$\overrightarrow{\mathsf{onion}}(\boldsymbol{sk}, m) := \mathsf{Enc}_{\mathsf{sk}_1}(\mathsf{Enc}_{\mathsf{sk}_2}(\ldots \mathsf{Enc}_{\mathsf{sk}_\ell}(m)\ldots)),$$

where $\boldsymbol{sk} := (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$ and m is the message.

Onion routing. Onion encryption is the backbone for onion routing, where a chain of servers route onion-encrypted messages, removing one layer of encryption at a time. Onion routing prevents any intermediate server from learning the plaintext message. We use the term **envelope** to describe the intermediate ciphertexts of an onion-encrypted message.

Boomerang encryption and routing. Onion routing works in one "direction:" routing envelopes *forward* through the chain of servers such that only the destination server learns the plaintext message. In contrast, boomerang routing is bidirectional: it applies onion encryption *twice*, once in the forward direction and once in reverse

(hence the name, *boomerang*). If we define reverse-onion encryption as:

$$\overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, m) := \mathsf{Enc}_{\mathsf{sk}_\ell}(\mathsf{Enc}_{\mathsf{sk}_{\ell-1}}(\ldots \mathsf{Enc}_{\mathsf{sk}_1}(m)\ldots)),$$

then boomerang encryption is defined as:

$$\overleftarrow{\mathsf{boomerang}}(\boldsymbol{sk}, \overrightarrow{m}, \overleftarrow{m}) := \overrightarrow{\mathsf{onion}}(\boldsymbol{sk}, \overrightarrow{m} \parallel \overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, \overleftarrow{m}))),$$

where $\overrightarrow{m}$ is the forward message and $\overleftarrow{m}$ is the returned message. We note that the $\overleftarrow{m}$ cannot be returned without first decrypting all envelopes in the forward onion chain.

Boomerang decryption as a proof-of-delivery. We make the following somewhat surprising observation: a successful boomerang *decryption* can be turned into a proof of message delivery when applied to onion routing.[1] Specifically, letting $\overleftarrow{m}$ (the returned message) be a random nonce acts as a message delivery receipt: it guarantees that every server along the path decrypted the correct onion layer and forwarded it to the next server. To see why this holds, observe that in order to successfully recover $\overleftarrow{m}$, all envelopes in the forward onion must be decrypted to recover $\overrightarrow{m} \parallel \overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, \overleftarrow{m})$. Therefore, message $\overrightarrow{m}$ must have been delivered to the server at the end of the path (i.e., the last envelope in the forward onion chain must have been decrypted). We formalize this in Appendix 5.2.3.

3.4.3 Anytrust key generation

ARTs signed by different groups must be indistinguishable. Therefore, each group needs to have the *same* public key to prevent leaking information (using different signing keys would leak which anytrust group signed the token). One simple (but flawed!) way to achieve this would be to give the same secret shares to all of the anytrust groups. Unfortunately, the failure of this approach is that an adversary controlling different servers across different groups can learn the entire set of (secret)

[1] Note that proof-of-delivery requires the underlying encryption scheme to be IND-CCA secure (such as Encrypt-then-MAC [11] with AES).

signing keys. (The adversary can corrupt different members in each group to obtain all s signing keys.)

Fortunately, this is the exact use case for *proactive* secret sharing [48] (and redistributable secret sharing [113]). The beautiful idea behind proactive secret sharing is that is is possible to generate *secret shares* of secret shares. Doing so generates a fresh set of secret shares encoding the *same* secret. The end result is that the anytrust groups have (1) a common public key but (2) *different secret shares* of the secret signing key x.

Generating ART signing keys in Trellis. One anytrust group, which we call the *main group*, runs a distributed key generation protocol [42, 55, 85] to generate secret shares of the signing key x corresponding to a public verification key tvk. The shares of x, denoted x_i where the ith group member holds x_i satisfy the property that $x = \sum_{i=1}^{s} x_i$. The challenge, as described above, is to efficiently generate shares x'_i, for each i, such that $x = \sum_{i=1}^{s} x'_i$ (specifically, we want to avoid *inefficient* multi-party computation or re-running distributed key generation again). Using the idea of proactive secret sharing, each group member uses a *verifiable* secret sharing scheme [89] to generate fresh secret shares of their share x_i, which we will call *sub-shares* of x_i and denote by $(x_i^{(1)}, \ldots, x_i^{(s)})$. Each set of sub-shares sums to one of the original shares, so $x_i = \sum_{j=1}^{s} x_i^{(j)}$. Then, one sub-share from each set is distributed to each member of the new group. This group member can sum the sub-shares to get a share of the original secret, since the sum of all of the sub-shares is equal to the sum of the shares. Specifically, the ith member in the new group computes: $x'_i = \sum_{j=1}^{s} x_j^{(i)}$. Observe that $x'_1 \ldots x'_s$ held by the members in the new group satisfy $x = \sum_{i=1}^{s} x'_i$, and each share is random given that one of the original group members is honest. This can be seen by flipping the order of summation: we have $\sum_{i=1}^{s} x'_i = \sum_{i=1}^{s} \left(\sum_{j=1}^{s} x_j^{(i)} \right) = \sum_{i=1}^{s} (x_i) = x$.

3.4.4 Verifiable secret sharing for Diffie-Hellman messages

A t-out-of-n verifiable secret sharing scheme (VSS) allows any subset of t shares to reconstruct the secret. Furthermore, every share is verifiable, meaning the share can be verified for consistency against the secret. In particular, we use the Feldman scheme [11, 39, 56]: Each server will act as the dealer for a VSS instance and create a random polynomial $P(x) = a_0 + a_1 x + \ldots a_{t-1} x^{t-1}$, where a_0 is the Diffie-Hellman secret. For each coefficient, the dealer publishes a binding commitment g^{a_i}, and note that g^{a_0} is the Diffie-Hellman public message. This dealer gives to each of the other servers a share $s_i = \alpha_i, P(\alpha_i)$, where α_i is a point on the polynomial. To avoid duplicate shares, it is standard to choose $\alpha_i = i$ for $i = 1 \ldots n$, for some public ordering of the servers. Anyone can verify a share is valid by checking that

$$(g^{a_0})(g^{a_1})^{\alpha_i}(g^{a_2})^{\alpha_i^2} \ldots (g^{a_{t-1}})^{\alpha_i^{t-1}} \stackrel{?}{=} g^{P(\alpha_i)}.$$

In particular, all servers will verify their share from the dealer, and blame the dealer by producing a signed message containing an incorrect share (or signed messages with equivocating commitments, etc.). In our case, the first commitment is the Diffie-Hellman public message g^{a_0}. Or, in other words, the commitment value is used as the Diffie-Hellman public message for encryption, and the dealer will be unable to decrypt messages if it uses a different value.

If the dealer is to be replaced in blame protocols, servers vote using their secret shares for the corresponding VSS instance. These shares are sent to the (randomly) chosen replacement, who tallies the votes by verifying each share it is given. With t (valid) shares, the replacement can interpolate the coefficients of the polynomial and recover the Diffie-Hellman secret, a_0. For any given α_i, the value $g^{P(\alpha_i)}$ can be computed using only the public commitments and is itself a binding commitment to $P(\alpha_i)$. There is only one possible share value, $P(\alpha_i)$, which is the discrete log of the publicly computable value $g^{P(\alpha_i)} = \prod_{j=0\ldots t-1}(g^{a_j})^{\alpha_i}$. Since there is only one value that matches the verification equation, it is not possible for a server to provide a fake share.

3.5 Putting things together

We decouple the expensive cryptographic operations from the lightweight ones (similarly to cMix [15]). During key generation and path establishment, all necessary cryptographic key material is distributed via the ARTs to servers on the path. During the mixing phase, messages are routed through the established paths. This is a lightweight procedure: each server only needs to decrypt and shuffle envelopes. The mixing phase can be repeated indefinitely (with many different messages) and does not involve the setup overhead of the path establishment phase, see Figure 3-2.

Key generation. Servers run the anytrust key generation protocol to generate shares of a common secret. The group outputs a public key, which can be computed from the commitments in Feldman secret sharing. Next, each server uses DHKeyGen and publishes the Diffie-Hellman public key to the public key infrastructure from which server can get TLS certificates for each other. Or more simple, this public key can simply be the TLS public key if it is compatible. Each server uses Feldman's verifiable secret sharing (VSS) scheme [39] (described above) to share the Diffie-Hellman secret, for state recovery purposes (Section 4.2.1).

ART generation. Each user is issued one ART per layer by a random anytrust group. Each user can only send (boomerang encrypted) messages through the network according to the path dictated by the ARTs. All invalid or duplicate envelopes are discarded by the honest servers. Servers that improperly route envelopes are blamed by the next honest server on the path (blame protocols are described in Chapter 4).

Routing tokens and path selection. During the path establishment protocol (described in further detail in Section 3.6), each server (anonymously) receives ARTs for the links it is responsible for. Each token determines the next server on a link (and can be checked with ART.NextServer) which allows each server to (1) determine where to send envelopes in the subsequent layer and (2) anonymously verify path adherence (without knowledge of the full path, only the local link).

Boomerang routing on a path. We use boomerang encryption to route envelopes through the mix-net. We assume that the boomerang encryption scheme is instanti-

ated using IND-CCA secure encryption (where the shared encryption key is derived from the Diffie-Hellman message in the ART payload). Each intermediate layer ciphertext (i.e., envelope) is signed and can be verified by the server using the signature verification key given in the corresponding ART payload. See Protocol 1 in **??** for details. This gives us *authenticated* boomerang routing which ensures that: (1) each server receives every envelope for each link and (2) all envelopes on a link are valid authenticated encryptions under the associated link public keys. Any failure of these two properties (e.g., failed decryption, bad signatures, or missing envelopes) triggers a blame protocol (described in Section 4.1.4).

Dummy envelopes. To prevent leaking how many messages are passed between servers in each layer (required to avoid leaking parts of the mixing permutation applied on the messages to the network adversary), each server batches the envelopes (intermediate boomerang ciphertexts) it sends to the destination server(s) in the next layer, padding the batch with "dummy" envelopes as needed. The dummy envelopes are then discarded by the receiving server(s).

Assigning blame. All communication between parties is additionally signed using a digital signature (DS) scheme for lightweight blaming of malicious parties. See Chapter 4 for details on our blame protocols.

3.6 Onion-routing on a pre-established path

The first protocol (that we formalize in Protocol 1) describes how servers route envelopes on a pre-established path. This protocol is used as a building block to route the forward and reverse onions of the boomerang encrypted messages. At each server, we peel off one layer of the onion, and then forward to the next destinations. We assume that we know these destinations because we have already received the ARTs during a previous path establishment round. The protocol summarizes what to do in the various failure cases, and which blame protocol to invoke accordingly.

Protocol 1: Onion routing on a pre-established path

This protocol is used by servers to route envelopes on paths consisting of pre-established links through ART tokens. The servers verify that envelopes are signed and on routes matching the tokens, and that all envelopes are accounted for.

Server state:

- List of digital signature public keys, one for each server, and the secret signing key for this server,
- list of ART tokens and corresponding information $(\mathsf{t}_\ell, A_\ell, \mathsf{vk}_\ell, \mathsf{vk}_{\ell+1})$ for each layer $\ell \in \{1, \ldots, L\}$,
- DH key agreement secret b,
- and set of redeemed tokens $T := \{\}$.

Server S_k input: Signed batches of envelopes $(\mathcal{B}_{j,k} := \{E_{i,j} \mid 1 \leq i \leq B\}, \sigma_{j,k,\ell})$ from each server $S_j \in \mathbb{S}$ for layer ℓ.
Server S_k output: Signed batches of envelopes $(\mathcal{B}_{k,j} := \{E_{i,j} \mid 1 \leq i \leq B\}, \sigma_{k,j,\ell+1})$ for each server $S_j \in \mathbb{S}$ for layer $\ell + 1$.

Procedure: Run once for each layer ℓ

1: **for** each incoming batch, from server S_j
   1: Verify $\sigma_{j,k,\ell}$ signs $\mathcal{B}_{j,k}$ with server S_j's public key.
   2: **for** each received non-dummy envelope $E_{i,j} \in \mathcal{B}_{j,k}$
      2.1: Parse $E_{i,j} = (\mathsf{vk}_\ell, c_\ell)$.
      2.2: Find the token and corresponding information $(\mathsf{t}_\ell, A_\ell, \mathsf{vk}_\ell, \mathsf{vk}_{\ell+1})$ in the server state that has (vk_ℓ).
         - If the token with vk_ℓ does not exist, or was delivered by the wrong server, blame with the protocol in Section 4.1.3.
      2.3: $\mathsf{sk}_i \leftarrow \mathsf{DHKeyAgree}(A_\ell, b)$.
      2.4: $\mathsf{Verify}(\mathsf{vk}_\ell, c_\ell)$. // Both for blame and upgrade to IND-CCA secure encryption
         - If verification fails, blame with the protocol in Section 4.1.3.
      2.5: $c_{\ell+1} \leftarrow \mathsf{Dec}_{\mathsf{sk}_i}(c_\ell)$.
      2.6: $S_{\ell+1} \leftarrow \mathsf{NextServer}(\mathsf{tvk}_\ell, \mathsf{t}_\ell)$.
      2.7: Add $(\mathsf{vk}_{\ell+1}, c_{\ell+1})$ to the batch $\mathcal{B}_{k,z}$ to send to $S_{\ell+1}$.
      2.8: Add t_ℓ to the list of redeemed tokens.
2: Check that there are no duplicates in T. If there exist duplicates, blame with the protocol in Section 4.1.2.
3: Check that $|T|$ is equal to the number of ARTs for layer ℓ. If not, blame with the protocol in Section 4.1.4.
4: **for** each server $S_j \in \mathbb{S}$
   1: Pad batch $\mathcal{B}_{k,j}$ with dummy envelopes to length B.
   2: Randomly permute $\mathcal{B}_{k,j}$.
   3: Compute $\sigma_{k,j,\ell+1}$ on $\mathcal{B}_{k,j}$ using sk_k.
   4: Send each batch and signature $(\mathcal{B}_{k,j}, \sigma_{k,j,\ell+1})$ to server S_j.

3.7 Inductive path extension

Authenticated onion routing works because all servers know which envelopes to expect from the routing information. This still leaves the challenge of how we get the routing information in the first place. The difficulty is that paths must be created in a way that (1) does not reveal the path to any party other than the user and (2) guarantees that all paths are uniformly random, even when generated by malicious users. Each server on the path must receive an ART associated with the path link but must not learn which user provided it. We solve this with an incremental approach: we inductively extend an existing path from one server to another while simultaneously forcing all users to choose *random* paths. That is, the user starts with a random path of length one, with the choice of this first server assigned by the first ART. This base case does not yet provide anonymity since the first server knows the user's identity. We then proceed by induction as follows. Given a path of length ℓ ending at server S_ℓ, the user extends the path to length $\ell + 1$ using boomerang routing through $S_1 \to S_2 \to \ldots \to S_\ell$ to communicate with the new server $S_{\ell+1}$. The user sends the routing information $S_{\ell+1}$ needs: the ART $\mathtt{t}_\ell$ for the link $(S_\ell \to S_{\ell+1})$, the ART $\mathtt{t}_{\ell+1}$ for the link $(S_{\ell+1} \to S_{\ell+2})$, and the associated Diffie-Hellman messages and signature verification keys. (The user also sends a signature on the ART $\mathtt{t}_{\ell+1}$ under the verification key $\mathtt{t}_\ell$ to prove that $\mathtt{t}_\ell$ and $\mathtt{t}_{\ell+1}$ are part of the same path.) Because the user communicates exclusively through the existing path and because ARTs are unlinkable, the path extension procedure reveals no information to any server. After receiving and verifying the tokens, $S_{\ell+1}$ routes the reverse boomerang envelopes backwards along the path. As each server verifies the authenticated encryption, it implicitly ensures that the user and all servers $S_\ell \to S_{\ell-1} \to \ldots \to S_1$ know that $S_{\ell+1}$ received (and validated) the tokens (if server S_ℓ does not do so then it is blamed by an honest server in a subsequent round). This process is repeated L times until the full path to S_L is established.

Anytrust group bookends. The Trellis mix-net consists of L mix-net layers "book-ended" between two layers of anytrust groups (see Figure 3-5). The anytrust groups

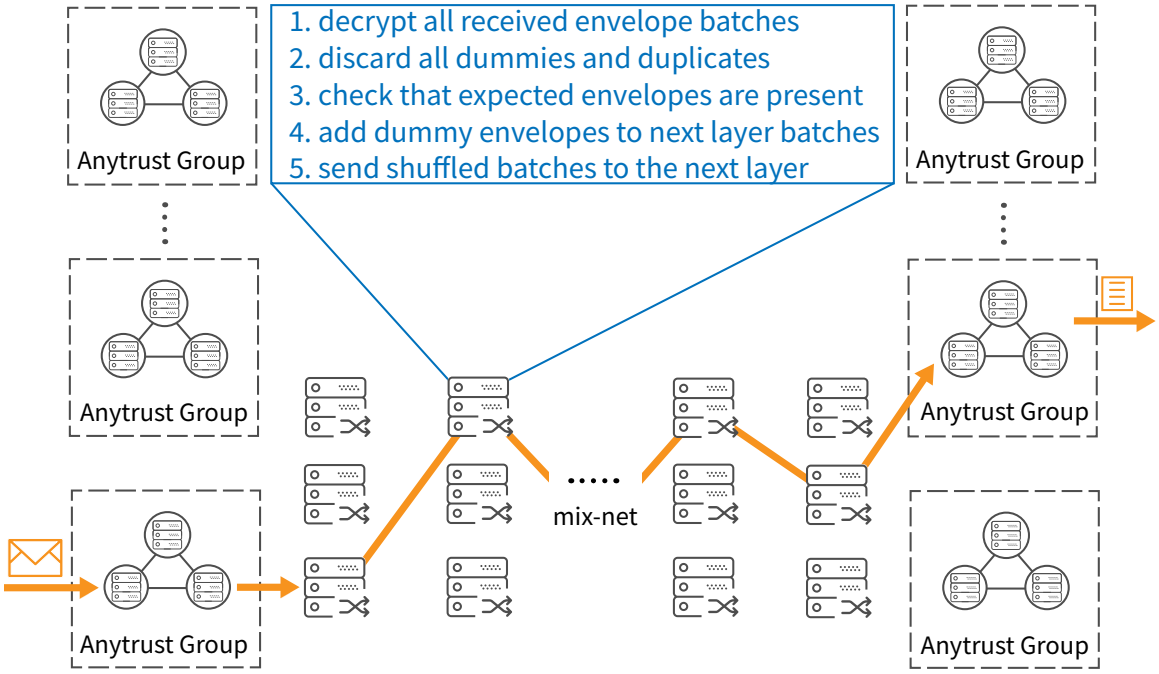

Figure 3-5: Detailed overview of the Trellis mix-net "bookended" by anytrust groups to prevent denial-of-service attacks at the entry and exit points. Users send boomerang-encrypted messages through a pre-selected random path in the network. The anytrust group at the last layer ensures that all messages get published to the bulletin board.

at the entry and exit layers ensure that the start and end of a mix-net path are always processed by an honest server, which prevents dropped messages and denial-of-service attacks. The anytrust group at the entry layer verifies the boomerang receipts on behalf of the user: When the boomerang receipt comes back, each member of the group can check the attached signature and ensure that the receipt was correctly decrypted. Additionally, the entry group stores the envelopes it is given so as to prevent denial-of-service attacks: If the first k servers in the mix-net are malicious and drop messages, the entry group can resend the envelopes after the adversarial servers are eliminated. The anytrust group at the exit layer outputs the messages to the world (the "broadcast" part of anonymous broadcast), ensuring that all messages make it out. During path establishment, the exit groups are at the "peak" of the boomerang and use a group decryption protocol to decrypt the Lth boomerang envelopes. In this protocol, each group verifies the ART for layer L and retains the included signature verification key. (Note that the keys are unlinkable to the start of the path because of the mixing

guarantees provided by the mix-net.) Then, during the repeated broadcast rounds, these exit groups check that one message is received for each of the verification keys they have, ensuring delivery of all messages.

Envelope generation. Users first engage in a protocol with anytrust groups to create ARTs, which are then used to construct envelopes for path establishment. The path establishment message contains two ART tokens and payloads—one for the incoming link and one for the outgoing link. These two ARTs are additionally binded together by the user with an included digital signature. This signature verifies with the key in the ART payload, and is used in the blame protocol of Section 4.2.3 to prove that the tokens are owned by the same user.

The users generate envelopes for L path establishment rounds and submit them to anytrust groups. Each of these anytrust groups is assigned a subset of users to manage, and each user provides their assigned anytrust group a signature verification key. These keys are used to verify the boomerang receipts on behalf of the user. To do so, for each key, the group checks that the there is a correctly signed receipt, and concludes that the receipts have been delivered if the signatures verify. See Protocol 2 for details on path establishment.

Protocol 2: Path extension (setup and users)

This protocol is used by the user and servers to extend an existing path of length ℓ to a new path of length $\ell+1$. This protocol is run simultaneously by all users in synchronous rounds. We assume that all messages to and from anytrust groups during ART signing are authenticated using a digital signature.

One-time setup for servers.
Each server generates and posts the public Diffie-Hellman message,

1: $B, b \leftarrow \mathsf{DHKeyGen}(\mathbb{G}, g)$.

2: Publish B to a publicly accessible bulletin board (or send to all users).

User-side ART generation and path initialization.
Each user generates ART tokens for all L layers. The ℓth token encapsulates a new Diffie-Hellman message and new signature verification key for the ℓth layer.

1: **for** $\ell = 0 \ldots (L+1)$**:**

1.1: $(\mathsf{vk}_\ell, \mathsf{ssk}_\ell) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$.

1.2: $(A_\ell, a_\ell) \leftarrow \mathsf{DHKeyGen}(\mathbb{G}, g)$.

1.3: $\mathsf{payload}_\ell := (S_\ell, A_\ell, \mathsf{vk}_\ell)$. // $S_0 := \perp$

1.4: $(\mathsf{st}_\ell, T_\ell) \leftarrow \mathsf{User}_0(\mathsf{tvk}_\ell, \mathsf{payload}_\ell)$.

1.5: Send T_ℓ to an anytrust group for signing; receive in return $W_{\ell,i}$ from the ith group member, for $i \in \{1, \ldots, s\}$.

1.6: $\mathsf{t}_\ell \leftarrow \mathsf{ART.User}_1(\mathsf{st}_\ell, W_{\ell,1}, \ldots, W_{\ell,s})$; $S_{\ell+1} \leftarrow \mathsf{ART.NextServer}(\mathsf{tvk}_\ell, \mathsf{t}_\ell)$.

1.7: Check the signature using $\mathsf{ART.Verify}(\mathsf{tvk}_\ell, S_{\ell+1}, \mathsf{payload}_\ell, \mathsf{t}_\ell)$. If no, blame with the protocol in Section 4.1.5.

1.8: $B_\ell \leftarrow$ the B Diffie-Hellman message published by server S_ℓ. // Skip on iteration $\ell = 0$.

1.9: $\mathsf{sk}_\ell \leftarrow \mathsf{DHKeyAgree}(B_\ell, a_\ell)$. // Key for forward boomerang.

1.10: $\mathsf{sk}'_\ell \leftarrow \mathsf{DHKeyAgree}(B_{\ell-1}, a_\ell)$ .// Key for reverse boomerang.

2: $\boldsymbol{sk} := (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell, \mathsf{sk}'_\ell, \ldots, \mathsf{sk}'_1)$ // Boomerang encryption keys.

3: **for** $\ell = 1 \ldots L$**:**

1.1: $r \xleftarrow{R} \{0,1\}^\kappa$. // Random nonce for receipt of delivery.

1.2: $\Sigma_\ell \leftarrow \mathsf{DS.Sign}(\mathsf{ssk}_0, r||\ell)$.

1.3: $\sigma_\ell \leftarrow \mathsf{DS.Sign}(\mathsf{ssk}_\ell, \mathsf{t}_\ell||\mathsf{t}_{\ell+1})$.

1.4: $c_{\ell,0} \leftarrow \mathsf{boomerang.Enc}(\boldsymbol{sk}, \mathsf{t}_\ell||\mathsf{t}_{\ell+1}||A_\ell||\mathsf{vk}_\ell||A_{\ell+1}||\mathsf{vk}_{\ell+1}||\sigma_\ell, r||\Sigma_\ell)$ and each intermediate envelope is signed against the corresponding vk_ℓ. If ℓ is L, then additionally encrypt the reverse onion with $\mathsf{DHKeyAgree}(\mathsf{gpk}, a_{L+1})$.

1.5: Submit $c_{\ell,0}$ and vk_0 to S_1 and an anytrust group. When all users have submitted, servers run path establishment for layer ℓ (Protocol 3).

4: The anytrust group checks that it receives a nonce r and signature Σ_ℓ corresponding to each user's signature verification key vk_0. If missing, blame with the protocol in Section 4.1.4. If the signature is invalid, blame with the protocol in Section 4.1.3.

Envelope routing. After setup, servers route envelopes to establish paths without requiring any further interaction from the user. The protocol is a variation of authenticated onion routing, described in Protocol 1. There are three main changes, which we summarize here:

1. At the peak of the boomerang, servers check that the tokens are valid, and store them for use in later rounds.
2. When routing onion envelopes in the reverse direction, encryptions must use the receiving server's key (i.e., the roles of the sender and receiver switch).
3. Envelopes are decrypted by an anytrust group during the final path establishment round (the Lth layer).

The formal details of the protocol are described in Protocol 3.

Reverse onion routing. The ART payload corresponds to a link between servers. All of the user's messages on the path link use the Diffie-Hellman public message and signature verification key from the payload. However, in reverse onion routing, we use the Diffie-Hellman public message from S_ℓ to encrypt, instead of the one from $S_{\ell+1}$. This is because S_ℓ is the destination of the envelope, and will be the one decrypting the message in the reverse direction.

Anytrust group decryption. The anytrust group uses a special group decryption protocol at the peak of the boomerang. The onion ciphertext key is formed using an anytrust group public key for the layer (created using proactive secret sharing) such that all anytrust groups have shares of the same secret key. The user's message is then encrypted under $(\mathsf{gpk}_L)^a$. To decrypt, we need to calculate $(g^a)^x$, where g^a is the Diffie-Hellman public message of the user, and x is the anytrust group secret key. We can do this by having each group member calculate $(g^a)^{x_i}$, where x_i is an additive share of x. Then the product $\prod_i (g^a)^{x_i} = (g^a)^{\sum_i x_i} = (g^a)^x$. Furthermore, we can delegate decryption of this message by sending the partial computations to server S_L.

Delivery is ensured by the anytrust group. Each anytrust group member checks the final ART t_L and records the signature verification key from the payload. They

only provide the partial computation if they receive a valid ART, and therefore the final receipt will guarantee that the signature verification key was delivered to all the members of the anytrust group. The honest member in this group will then use the verification key to ensure messages from the user arrive and are published at the end of the broadcast round.

Protocol 3: Path extension (servers)

Path establishment for layer ℓ.
Servers apply authenticated onion routing (Protocol 1) to route and decrypt $\overleftarrow{\mathsf{onion}}$ to S_ℓ (the path is pre-established through $\ell - 1$).

Each server (denoted by S_ℓ):

1: Run Protocol 1 with the following modifications:

- After step 2.5,
  1: $\overrightarrow{m} \leftarrow \mathsf{Dec}_{\mathsf{sk}_i}(c_\ell)$ and parse $\overrightarrow{m} = \mathsf{t}_\ell||\mathsf{t}_{\ell+1}||A_\ell||\mathsf{vk}_\ell||A_{\ell+1}||\mathsf{vk}_{\ell+1}||\sigma_\ell||c'_\ell$.
  2: Check the following signatures:
     - $\mathsf{ART.Verify}(\mathsf{tvk}_\ell, S_j, S_\ell, (A_\ell, \mathsf{vk}_\ell), \mathsf{t}_\ell)$,
     - $\mathsf{ART.Verify}(\mathsf{tvk}_{\ell+1}, S_\ell, S_{\ell+1}, (A_{\ell+1}, \mathsf{vk}_{\ell+1}), \mathsf{t}_{\ell+1})$, and
     - $\mathsf{DS.Verify}(\mathsf{vk}_\ell, \mathsf{t}_\ell||\mathsf{t}_{\ell+1}, \sigma_\ell)$.

     If any check fails, blame with the protocol in Section 4.1.3.
  3: Store $(\mathsf{t}_\ell, A_\ell, \mathsf{vk}_\ell, \mathsf{vk}_{\ell+1})$ in the server state list for layer ℓ.
- If $\ell = L$, then also:
  1: Send $(A_{L+1}, \mathsf{vk}_{L+1}, \mathsf{t}_{L+1})$ to each member of an anytrust group. Each group member:
     - Checks $\mathsf{ART.Verify}(\mathsf{tvk}_{L+1}, S_{L+1}, (S_L, A_{L+1}, \mathsf{vk}_{L+1}), \mathsf{t}_{L+1}) = \mathsf{yes}$. If Verify outputs no, blame with the protocol Section 4.1.3.
     - Stores vk_{L+1} for future use.
     - Outputs $(A_{L+1})^{x_j}$ where x_j is the member's share of the anytrust group secret key.
  2: Compute the product $\prod_j (A_{L+1})^{x_j} = \mathsf{DHKeyAgree}(\mathsf{gpk}, \mathsf{sk}_{L+1})$ to decrypt one layer of c'_ℓ.
  3: If decryption fails, blame as described in Section 4.1.6.
- For step 2.7, instead add $(\mathsf{vk}_\ell, c'_\ell)$ to the batch for S_j (the server that the envelope came from).

2: **All servers:** Apply authenticated onion routing (Protocol 1) in reverse to route and decrypt c'_ℓ back to the user.

Why path establishment works. Both the incremental path extension and the

boomerang message receipts are required to prevent dropped message attacks. The incremental approach to path establishment enables immediate blame assignment and prevents errors from propagating. The message receipt is designed to allow servers on the existing path to assign blame on a user's behalf if the new server deviates from protocol. Specifically, each server expects to receive an envelope for each established key, in both the forward and reverse direction. This gives rise to a simple blame protocol in the case of a dropped envelope: blame the previous server. The previous server, if innocent, should have reported the envelope missing during the last synchronous layer; because it did not do so, it must be malicious. See Chapter 4 for details.

3.8 Broadcast round protocol

Once we have established paths, we can use authenticated onion routing to quickly route messages. These established random routes guarantee that all messages are mixed before reaching the final layer (see mixing analysis in the next section). For each round, messages are of fixed size (e.g., 10kB), and all users must submit a message of that size, padding if necessary. Users with larger messages can split their broadcasts across several rounds. Note that path establishment does not depend on the message size or the number of broadcast rounds that come later, enabling us to amortize the cost of establishing paths and to mix large messages.

In each broadcast round, the user signs its message and concatenates this signature. Both the message and signature are encrypted in the onion encryption. This signature is signed with the key corresponding to the last layer, such that the anytrust group has the corresponding verification key. When messages are routed through the mixnet, these anytrust groups will check for the presence of a correctly signed message under each signature, which will allow them to detect and ensure all messages are output. Because the anytrust group checks the message, it is not necessary to give boomerang receipts for proof of delivery — delivery is already guaranteed.

Protocol 4: Broadcast round

The broadcast protocol is repeated many times to transfer large messages, independent of the setup time. We assume that signatures include the current round number $\mathbf{r}$ when there are many rounds, shown explicitly here for completeness.

User

1: $\sigma_{L+1} = \mathsf{DS.Sign}(\mathsf{sk}_{L+1}, \mathbf{r}||m)$.
2: $c_0 \leftarrow \overrightarrow{\mathsf{onion}}(\mathsf{sk}_{1\ldots L}, m||\sigma_{L+1})$.
3: Send c_0 to S_1.

Servers

1: Servers apply authenticated onion routing (Protocol 1) using the pre-established paths to route and decrypt c_0 to S_L.
2: The last server, S_L, sends each $(\mathsf{vk}_{L+1}, c_{L+1})$ to each member of the anytrust group handling vk_{L+1}.

Anytrust group members

1: **for each received message** $(\mathsf{vk}_{L+1}, c_{L+1})$**:**
   1: Parse $m, \sigma_{L+1} = c_{L+1}$.
   2: Check $\mathsf{DS.Verify}(\mathsf{vk}_{L+1}, \mathbf{r}||m, \sigma_{L+1})$. If the signature is incorrect, blame with the protocol in Section 4.1.3.
2: Check that each verification key vk_{L+1} was used exactly once. If not, blame with the protocol in Section 4.1.4.
3: Forward messages to the final destination.

3.9 Number of layers required

In this section we provide the full analysis for the number of layers required to achieve a random permutation on the messages with variation distance at most ϵ, for any small ϵ (e.g., $\epsilon < 2^{-64}$).

Theorem 1. Let M be the number of messages being shuffled (equivalent to the number of users) and $N \geq 2$ be the number of servers in the network. Fix $0 \leq f < 1$ to be a fraction of corrupted servers and let $\epsilon < 1$ be a statistical security parameter. If each message is routed through a random server L times and it holds that:

$$L \geq \left\lceil \frac{-\log \epsilon + \log M + \log(2\sqrt{4f - 3f^2}) + \log(2 - f + \sqrt{4f - 3f^2})}{\log(\frac{2}{f + \sqrt{4f - 3f^2}})} \right\rceil + 1,$$

then the distrubtion of messages output by servers in the Lth layer is "ϵ-close" to a uniformly random permutation of all messages (the output distribution has variation distance at most ϵ to the distribution of random permutations on all messages).

Proof. Without loss of generality, we assume that (1) the servers are numbered $1, 2, \ldots N$ with the honest servers first and that (2) each message is represented as a card labeled from 1 to M.

We construct a deck of cards corresponding to the ordering from the first message in server 1 to the last message in server N and consider how the deck is shuffled as we simulate how the messages move through the servers. Each state of the network can be thought of as a $M \times 2$ matrix. In this matrix, each row corresponds to the message (card) labeled i. The first column indicates the current position of the ith message in the deck. The second column indicates the server currently holding the ith message. Messages move between the servers at random[2] and we track how the deck of cards (messages) is shuffled from the adversary's view. Crucially, we must consider how this view changes when cards (messages) are located in adversarial servers. We model this process as follows.

1. Choose the next server for each message uniformly at random. This corresponds to a vector of length M with entries chosen uniformly at random from $\mathbb{Z}_N$.
2. All messages that start and end in an honest server in one time step are shuffled together. We represent this by selecting these cards out of the deck, shuffling them, and placing them back on the top of the card deck. We use this to model the following behaviors:
   - Cards that travel between honest servers are shuffled in the view of the adversary, and placed on the top of the deck (the top of the deck consists of the honest servers).
   - The remaining cards are pushed down the deck, but do not change ordering. The only shuffling we consider is during honest-to-honest transitions. All of the remaining cards stay in the same relative order.

[2]In Trellis this is guaranteed by the verifiable randomness of the ART tokens which chooses the next server uniformly at random.

- Some cards are pushed from the honest section of the deck to the adversarial, modeling honest-to-adversary transitions.
- Some cards remain in order in the adversary section of the deck, and correspond to adversary-to-adversary transitions.
- Finally, there are adversary-to-honest transitions, which we ignore for simplicity. [3]

3. Finally, we output the new state, consisting of the new arrangement of cards in the (new locations of the messages).

We now define a Markov chain coupling (see the excellent exposition of Klappenecker [57]) by using the randomness in step one to run two Markov chains in parallel [8]. We select the next server position for each message in both chains to be the same, so we will transition from honest to honest servers at the same time (after the first step). We choose the shuffles in step two so that each of the shuffled cards (messages) we place at the top of the deck are given the same position in both chains (all permutations of the selected set are equally likely). We consider cards that are at the same server and position in the deck at a given timestep to be *coupled*. Coupled cards have the same state row in both chains, and will continue to have the same state indefinitely given that they are either (1) pushed down the stack together or (2) both selected and shuffled to the same position.

We start one of the chains in the uniform distribution of $\mathbb{Z}_M^2$, and the other chain in an arbitrary starting distribution chosen by the adversary. We know, however, that the total variation distance between the two distributions must decrease over time as when all cards have coupled both chains will forever be in the same state, and from then on must have the same distribution. In particular, the uniform distribution is stationary for this chain, so any distribution will eventually become uniform.

The time to coupling (and hence the number of layers required) can be derived as follows. For each card, the probability the card does not couple in the first ℓ steps

[3] Shuffling within an honest server does not significantly decrease the number of layers required (and greatly increases the complexity of our proof) which is why we can ignore the details of this transition.

is the probability it is not in consecutive honest servers. The probability of being in an honest server is $1 - f$ (recall that we select the next server uniformly among all of the servers). This can be modeled as flipping a coin which comes up heads with probability $(1 - f)$ a total of ℓ times and asking what the probability is of *not* coming up heads twice in a row. We can solve for this probability using a linear recurrence relation. Then, we can union bound the probability that *any* message has not coupled, and require this to be less than $\epsilon = 2^{-\kappa}$. The dominant term of the solution[4] yields at least ℓ layers, where ℓ is

$$\frac{\kappa \log 2 + \log M + \log(2\sqrt{4f - 3f^2}) + \log(2 - f + \sqrt{4f - 3f^2})}{\log(\frac{2}{f+\sqrt{4f-3f^2}})}.$$

Accounting for the first step, the number of layers required is $L = \lceil \ell + 1 \rceil$. □

3.10 Theoretical efficiency

3.10.1 Computation

Each server processes the envelopes assigned to it by the anonymous routing tokens, which assign tokens evenly and randomly across all N servers. The expected number of envelopes processed by each server is thus $\frac{M}{N}$. We can use a binomial bound [6] to upper bound the number of messages processed by a server, with high probability. This bound also serves to calculate the number of dummy envelopes a server must send, which we calculate to be $O(\sqrt{M})$. There are $O(N)$ anytrust groups (anytrust group size depends only on f), and therefore each group processes $O(M/N)$ messages. By Section 3.9, we require $L \propto O(\log(M))$ layers. Note that envelope size is linear in the number of layers, as each layer adds the overhead of authentication. In total, the time (and bandwidth) required to route $O(M/N)$ messages of size $O(\log M)$ through $O(\log M)$ layers is $O(\frac{M \log^2 M}{N}) = \tilde{O}(\frac{M}{N})$. Since path establishment repeats L times, the time (and bandwidth) for path establishment is $O(\frac{M \log^3 M}{N}) = \tilde{O}(\frac{M}{N})$.

[4]See `https://puzzling.stackexchange.com/questions/29256/no-two-heads-in-a-row` for how to solve the linear recurrence relation (with unbiased coins).

3.10.2 Bandwidth

Similarly, we can derive a Chernoff bound on the number of messages traveling along a link against the expectation $\frac{M}{N^2}$ for each of the N^2 links. We set this Chernoff bound to be B, the total number of message to be sent on a link when dummies are added. Since each server has N outgoing links, the bandwidth costs are also $O(\frac{M \log(M)^2}{N})$ for broadcast rounds and $O(\frac{M \log(M)^3}{N})$ for path establishment. Note, however, that there is a constant factor of N^2 that should be added to these asymptotics. This is because we must always send at least one dummy envelope, even if the expectation is small. Because of this, horizontal scalability will end when M approaches N^2.

Chapter 4

Robustness

If all parties behave honestly, then all envelopes are present and well-formed. In this case, boomerang routing and mixing analysis (Section 3.9) ensure that all messages are delivered and mixed (see security analysis; Chapter 5). However, an adversarial user or server may deviate from the protocol in arbitrary ways. To prevent system disruption, or worse, user deanonymization following active attacks, each (honest) server is expected to verify all envelopes it receives for layer ℓ *before* sending the shuffled batch of envelopes to layer ℓ+1. If any of the checks do not pass, the server must notify (all) the other servers before the protocol finishes layer ℓ (note that this server will not send messages for ℓ+1 until the error is resolved, and so layer ℓ should not complete). This prevents errors from propagating through honest servers; honest servers blame immediately in the layer where a check fails and recovery protocols are initiated to eliminate the blamed server(s).

Blaming without deanonymization. We make an important observation which allows us to maintain the anonymity of honest users when blaming malicious servers: if we can show that at least one of the two servers on a link misbehaved, then we can reveal the envelopes along that link *without* compromising user anonymity. We formalize this in Claim 2.

Claim 1 (Malicious links provide no anonymity)**.** Links between two servers, where at least one of the servers is an adversary, can be revealed without compromising

anonymity.

Proof. Consider the view of an adversarial server on a link consisting of at least one malicious server. The adversary knows all encrypted envelopes on the link. Revealing any of this information (to all of the servers) does not further the adversary's knowledge. Our analysis of Section 3.9 relies only on links between pairs of *honest* servers for this reason. □

4.1 Blame protocols

In order to illustrate our blame protocols, we need to provide more details about what exactly is sent between servers. First, all signatures sign the round, layer, source server (which may not be the signer), destination server, protocol, length, and direction of travel. This will allow anyone to publicly verify which step of which protocol a message corresponds to. Second, servers sign the batch of envelopes and corresponding ART keys. Signing the concatenation of the batch ensures that no server can equivocate on the number of envelopes it sent for a given round and layer. These signatures can use any standard public-key signature scheme [11, 56].

We assume that all blame messages are sent to all servers (e.g., using a standard gossip protocol [65, 83]). If we assume an honest majority of servers ($f < 0.5$; see Section 4.2.2 for why this assumption is without loss of generality), the majority vote assigns blame and selects a replacement server. In Chapter 5, we cover how blame protocols support our delivery guarantees. In short, we ensure that no honest server or honest user can ever be blamed through a blame protocol, and thus only adversaries are blamed. The blame protocols ensure that the tokens, signatures, and envelopes are present and well-formed, which will imply successful delivery.

If all parties behave honestly, then all envelopes are present and well formed. owever, an adversarial user or server may deviate from the protocol in arbitrary ways. To prevent system disruption, or worse, user deanonymization following active attacks, each (honest) server is expected to verify all envelopes it receives for layer ℓ *before* sending the shuffled batch of envelopes to layer $\ell+1$. If any of the checks do

not pass, the server must notify (all) the other servers before the protocol finishes layer ℓ (note that this server will not send messages for $\ell+1$ until the error is resolved, and so layer ℓ should not complete). This prevents errors from propagating through honest servers; honest servers blame immediately in the layer where a check fails and recovery protocols are initiated to eliminate the blamed server(s).

Blaming without deanonymization. We make an important observation which allows us to maintain the anonymity of honest users when blaming malicious servers: if we can show that at least one of the two servers on a link misbehaved, then we can reveal the envelopes along that link *without* compromising user anonymity. We formalize this in Claim 2.

Claim 2 (Malicious links provide no anonymity)**.** Links between two servers, where at least one of the servers is an adversary, can be revealed without compromising anonymity.

Proof. Consider the view of an adversarial server on a link consisting of at least one malicious server. The adversary knows all encrypted envelopes on the link. Revealing any of this information (to all of the servers) does not further the adversary's knowledge. Our analysis of Section 3.9 relies only on links between pairs of *honest* servers for this reason. □

4.1.1 Arbitration protocol

Our first blame protocol deals with a missing or incorrect server signature of the concatenated envelopes. The arbitration protocol begins when a server $S_{\ell+1}$ reports that the signature given by S_ℓ is missing (including if no envelopes were sent at all) or incorrect. In this case, we have either that S_ℓ is adversarial or $S_{\ell+1}$ is maliciously initiating the blame protocol. An anytrust group randomly chooses an arbitrator from the other servers (the honest member will contribute true randomness and ensure the selection is random). We then require S_ℓ to send messages to the arbitrator, denoted $A_{\ell,\ell+1}$, who forwards them to $S_{\ell+1}$. The arbitrator also checks that the signature on the batch is correct (using S_ℓ's public verification key). If messages are delivered

successfully (from S_ℓ to $S_{\ell+1}$) through $A_{\ell,\ell+1}$, then $A_{\ell,\ell+1}$ continues to arbitrate for future rounds. The performance of Trellis is only minimally impacted by arbitration, as it simply adds an extra forwarding and signature check for one layer.

If the arbitrator itself has a dispute with S_ℓ or $S_{\ell+1}$, then it can determine which server is honest from its perspective. When the arbitrator is honest, it will never have a dispute with the honest server. Once an arbitrator decides who the malicious server is (because it has a dispute), a new arbitrator is randomly selected by an anytrust group. If there is an honest majority of servers, then there will be more disputes involving the dishonest server than there are involving the honest server. For a dishonest majority, we can track all server disputes with a trust graph [108]: a graph in which each edge represents if two servers trust each other. Although the initial disagreement breaks only one trust graph link, we can choose arbitrators who trust both parties. Then, each dispute between the arbitrator and a party breaks another trust graph link. Wan et al. [108] prove that such an algorithm eventually converges to the honest clique, who can then reform without the troublesome members.

Because of the synchronous communication assumption, messages are always delivered between honest parties in a timely manner, and so the arbitration protocol is only invoked with adversaries. Arbitration can introduce a maximum overhead of a factor of $2\times$, as each message is forwarded one extra time.

4.1.2 Duplicate envelopes

There should be exactly one message per ART per round and layer. A server can blame a user by producing as evidence two different signed envelopes for the same round and layer. This evidence is submitted to all of the servers, who vote to remove the user if the evidence is valid (or to remove the server if the evidence is invalid). We set $N(1-f)$ as the minimum threshold of votes required to remove a server, as described in Section 4.2.1. All of the protocols below follow this format: a party will produce some publicly verifiable evidence, which the servers will check and vote on accordingly.

4.1.3 Envelope decryption failure

If an envelope fails to decrypt, then a server $S_{\ell+1}$ can request that the previous server S_ℓ be blamed. This means that we are in one of three cases:

1. a malicious user sent an ill-formed envelope,
2. S_ℓ tampered with the envelope, or
3. $S_{\ell+1}$ is maliciously invoking the blame protocol.

Since the potentially malicious user, S_ℓ, and $S_{\ell+1}$ all know both the envelope and the verifying ART token, $S_{\ell+1}$ can reveal these to prove the envelope is ill-formed (in a similar argument to Claim 2). In response, S_ℓ may claim it never sent the ill-formed envelope and token. It then requests that $S_{\ell+1}$ reveal the signed concatenation of envelopes on the link to prove that the ill-formed envelope and ART were sent by S_ℓ. In this case, either S_ℓ or $S_{\ell+1}$ is blamed.

Otherwise, S_ℓ agrees that the envelope was ill-formed with respect to the corresponding ART, and we won't blame $S_{\ell+1}$. We still need to decide if the user sent the ill-formed envelope, or S_ℓ tampered with it. If S_ℓ is honest, then it will be able to present a proof that it acted correctly. This proof involves revealing each step taken by S_ℓ and proving that each step was executed correctly. To prove this, S_ℓ first reveals the ART $\mathtt{t}_\ell$ and the encapsulated payload (containing a partial Diffie-Hellman message and verification key). Using a discrete log equivalence proof [28, 50, 58], S_ℓ proves correct decryption (without revealing its long-term secret key). Finally, it provides the user's signature on $\mathtt{t}_{\ell+1}$ under the key from $\mathtt{t}_\ell$ to show that $\mathtt{t}_{\ell+1}$ is the correct next token. With these elements the other servers can verify the following:

- S_ℓ received an envelope that was well-formed (signed), and therefore S_ℓ was not required to blame earlier.
- The decryption of said envelope was computed correctly, but the output of the decryption was an ill-formed envelope.
- The keys are correct and came from the user who created both of the ARTs, as $\mathtt{t}_\ell$ signs the payload.

With this information, it can only be that the user is to blame, and we apply the

protocol in Section 4.2.3 to deanonymize and remove the malicious user. This well-formed envelope is signed (anonymously) by the user under some key vk_ℓ, and so could only have been created by them. Similarly, the token $\mathsf{t}_{\ell+1}$ is signed by vk_ℓ and so is the correct next token, and the correct $\mathsf{vk}_{\ell+1}$. Therefore, the server has produced a proof that the decryption was performed correctly, but the decryption output is ill-formed.

4.1.4 Missing envelope

Each server expects to receive one envelope corresponding to each ART pair it received. In the event that an envelope is missing, $S_{\ell+1}$ blames server S_ℓ for not sending the envelope (and not reporting the envelope missing in the previous layer). We have two cases to consider:

- $S_{\ell+1}$ is maliciously invoking the blame protocol, or
- S_ℓ is malicious and dropped an envelope.

Again, by Claim 2, $S_{\ell+1}$ can reveal the signed concatenation of the envelopes on link $(S_\ell \to S_{\ell+1})$. Specifically, $S_{\ell+1}$ provides the signed batch (with the missing envelope) that it claims to have received from S_ℓ in addition to a signed batch from a previous round showing that S_ℓ sent an envelope with the ART in question. We can then apply the protocol described in Section 4.2.1 to remove the offending server.

Remark 1 (An edge case)**.** At the peak of the boomerang, note the pattern of sending $(S_\ell \to S_{\ell+1} \to S_\ell)$. In this case, if S_ℓ blames $S_{\ell+1}$ for dropping the receipt on $(S_{\ell+1} \to S_\ell)$, then $S_{\ell+1}$ can dispute that S_ℓ never sent the corresponding ART by presenting the signed concatenation from $(S_\ell \to S_{\ell+1})$.

4.1.5 Invalid ART signature

If one of the signers in an anytrust group incorrectly signs the ART during the signing protocol described in Section 3.4.1, the user receives an invalid signature (which it locally determines is invalid using $\mathsf{ART.Verify}$). In this case, the user can request that each signer (publicly) proves it computed the partial signature correctly. In our

construction, each signer can do so efficiently by providing a zero-knowledge proof of discrete log equivalence [28, 50, 58], proving that $W_{\ell,i}$ was calculated correctly with respect to g^{x_i}. Any signer that does not comply is eliminated using the protocol of Section 4.2.1. Note in contrast to [28], we only require a zero-knowledge proof when a signed misbehaves, and we will blame them accordingly.

4.1.6 Failure of decryption

Similarly, if the group decryption fails, meaning that the signature on the decrypted message is invalid, we can ask each group member to prove they decrypted correctly in zero-knowledge. Each group member proves correct computation using a non-interactive zero-knowledge discrete log equivalence proof [28] to show that they computed their partial share correctly. Specifically, the group member shows that the partial decryption $(A_{L+1})^{x_i}$ and the commitment to the secret share g^{x_i} have the same exponent (with respect to A_{L+1} and g). This proof is lightweight, but note that this proof is only required when a failure occurs.

4.2 Removal and recovery protocols

Once we have identified the adversarial party, we can proceed to remove them from the system. To remove adversarial servers, we ensure that each server's secret keys are t-out-of-n shared, using a verifiable secret sharing [89] (VSS) scheme, with $t > Nf$ at setup time (see Section 3.4.4 for details). Blame decisions are made as a group, using the t-out-of-n secret key shares as votes. Using the shares allows the server receiving t (or more) shares (i.e., votes) to recover the offending server's secrets and replace it.

4.2.1 Server removal and state recovery

When servers vote to blame a malicious server, a replacement server is chosen randomly from among the remaining servers by an anytrust group (as the arbitrator was in Section 4.1.1). Each server sends its share of the blamed server's secret key sk to

the replacement server, who can use the VSS scheme to reconstruct sk if at least t servers agree that the server is at fault. Next, to reconstruct the routing information, the ARTs and corresponding information can be resent by the other servers. We note that the replacement server may additionally need to show that it did not receive an ART during this replay and pass blame for a missing message accordingly, which it can do using the signature of the replay batch. With the keys, the replacement can decrypt and forward the messages encrypted to the old server. Note that the replacement server does not recover the batch signature key. Instead, it signs batches with a new signing key which lets other servers know that envelopes came from the replacement rather than the eliminated server.

This key recovery is only necessary to finish the current round, as we need to decrypt message that have already been encrypted. In future rounds, the replacement can post a new Diffie-Hellman public message that users will encrypt with, without changing the ARTs or paths.

Remark 2 (Server churn)**.** In the event of server churn, where an (honest or adversarial) server leaves the network and stops responding, we can also apply the state recovery protocol to select a replacement and continue.

4.2.2 Server removal and state recovery with a dishonest majority

If there is a dishonest majority, then the vote described in Section 4.2.1 can deadlock, with neither side meeting the threshold required to find a replacement. However, note that all honest servers will vote together, and only adversaries will vote incorrectly. If we partition into a new instance along the voting lines, then the partition with the honest servers (who all voted together) will have a higher fraction of honest servers than before. If we let $h := 1 - f$ be the fraction of honest servers, then we will require no more than $\lfloor 1/h \rfloor - 1$ restarts after voting deadlocks in order to achieve an honest majority. For example, if $1/2 < f < 2/3$, we will need at most 1 restart. This is because in order to deadlock the vote, at least $1/3$ of the servers must vote incorrectly

(all servers that vote incorrectly are adversaries), who will form one partition. The other partition will then consists of at most 2/3 of the servers, including all of the (at least) 1/3 honest servers, giving it an honest majority. Since the evidence is publicly verifiable, it can also be verified by the users, who can decide which partition to join. The honest users will evaluate the evidence the same as the honest servers, and so they will choose to send messages to the honest partition.

4.2.3 Blocking malicious users

Once a user has been blamed, we can revoke their keys by having each server on the user's path reveal the user's ARTs. Specifically, a server can produce both of the user's ARTs and the signature binding them together. Then, we can repeat this for each server on the path (by following the identities committed in the ARTs), to remove all of the user's tokens from the servers' lists. We assume that the set of users is fixed, as users who join late are inherently hard to anonymize due to statistical disclosure attacks (see Section 3.3). Users that are late must wait until the next restart of the protocol to join.

4.3 Other considerations

4.3.1 Failure of an honest server in an anytrust groups

When a server in an anytrust group fails, it takes with it its share of the anytrust group secret key. In this case, the work of the anytrust group should be redistributed to the remaining groups. In particular, it is possible there is an anytrust group with exactly one honest server. If the only honest server fails, then the group is no longer an "anytrust" group and should no longer be used. Rather, the anytrust group should be replaced with a new group. There are three uses of anytrust groups in Trellis, each of which can adapt as follows:

- Token generation: Users should use a different anytrust group to generate tokens.
- Message submission: The remaining members of the group should redistribute

the messages to the other groups. The new group checks the signatures on the messages to make sure they are valid.

- Final signature check: The remaining members of the group should redistributed the final ARTs and signature verification keys to the other groups. The new group checks the tokens to make sure they are valid.

To ensure all tokens will be accounted for in the new group, each anytrust group should publish the count of tokens. The honest server will ensure this published number is accurate. If there is an equivocation, we can ask the server claiming the larger value to back up its claim by producing the required number of valid tokens. Then, when tokens are reassigned, the new group checks that it received the correct quantity of tokens. Tokens are hashed to assign anytrust groups for the purpose of load balancing and an efficient check for deduplication between groups. Since the group has changed, this hash function no longer maps correctly. Instead, using a consistent hash function [54] would allow redistribution in the event of failure to be as even as possible, and allow servers to efficiently verify tokens are not duplicated between groups.

4.3.2 More efficient blame

Instead of broadcasting blame messages to all servers, we can broadcast to an anytrust group. The honest server in the anytrust group will correctly determine who is at fault, but the anytrust group might equivocate within itself because an adversarial member disagrees. Furthermore, note that the anytrust group could a dishonest majority, and so the group cannot make decisions simply by voting. Simply put, there is a fast path when the entire anytrust group agrees, and a slow path when they do not. However, we can make an incentive for the adversary to blame correctly by having each member of the group make their blame determination public. We still have to have all servers evaluate the evidence and decide, but now we can also blame all the members of the anytrust group who determined incorrectly. Assuming we choose an anytrust group disjoint from the parties involved in the initial disagreement, we can

ensure that an additional malicious server is removed during each blame protocol. Alternatively, if the adversary votes to blame itself, we reap the efficiency gains of only needing to have an anytrust group adjudicate. Either way is an improvment for the performance of the system.

Chapter 5

Security

In this section, we analyze the security properties of Trellis. We show that (1) messages are mixed and no metadata is leaked in the process, (2) messages are guaranteed to be delivered, and (3) blame protocols eliminate adversarial parties—users or servers—without compromising anonymity.

5.1 System security

Claim 3 (Path establishment)**.** All honest users establish a random path through the layers and each server on the path obtains: a decryption key and a signature verification key.

Proof. Boomerang encryption verifies that the anonymous routing tokens are delivered to each server along the path (see Claim 9 in Appendix 5.2.3. The blame and recovery protocols (Chapter 4) ensure this process completes. In particular, note that the user establishes a key with the final anytrust group. □

Claim 4 (Path adherence)**.** All messages submitted by honest users are routed through all honest servers along the chosen path determined by the ARTs.

Proof. Each honest user follows the ART signing protocol and obtains valid ARTs for each link. Suppose, towards contradiction, that the envelope sent by the user does **not** travel through an honest server, say S_ℓ, as dictated by the ART on the

ℓth layer. The presence of a correctly signed message in the final anytrust group at the last layer implies that all preceding onion layers were decrypted correctly (see Lemma 1 in Appendix 5.2.3). In turn, this means that S_ℓ decrypted a layer, which is a contradiction. □

Claim 5 (Metadata-private shuffling)**.** All envelopes sent between two honest servers on a link appear indistinguishable to an external observer.

Proof. The dummy messages ensure that each batch of envelopes exchanged between two honest servers is of fixed size B (and TLS prevents the network adversary from seeing the contents of the batch). Crucially, no malicious server or user can overload an honest-honest link so that there are more than B envelopes. This holds because (1) the envelopes expected by each server are uniquely associated with ARTs provided during path establishment and (2) no honest server would forward any envelope not uniquely associated with an ART. By (1) and (2), the size of the batch sent to the second honest server is always B (padded with dummies as needed). □

Claim 6 (user anonymity)**.** All messages contributed by honest users are mixed, resulting in a random permutation of all messages being output to the public bulletin board.

Proof. We combine Claim 4 and Claim 5. Since all honest messages travel through all honest servers, they travel through all honest-honest links. Since the messages travel along honest-honest links, these messages are shuffled. Then, by Section 3.9 all honest messages become unlinkable with their senders as the total variation distance of the applied permutation is ϵ-close to a truly random permutation of all messages. □

Claim 7. An honest user is never blamed as a result of any blame protocol described in Chapter 4, except with negligible probability in the computational security parameter λ.

Proof. A user is only blamed if either (1) there exist two envelopes for the same round and layer (Chapter 4) or (2) if there is an ill-formed envelope at some layer ℓ (Section 4.1.3). Because an honest user's envelopes are always well-formed, (1)

and (2) do not apply to honest users. Furthermore, a user cannot be blamed by a malicious invocation of Section 4.2.3 (user removal) because each ART used to trace back the path to the is signed by the verification key in the next ART, and an honest user would never sign a different user's ART. Hence, if an honest user is blamed then a malicious party was capable of forging the honest user's signature, which can only happen with negligible probability. □

Claim 8. An honest server is never blamed as a result of a blame protocol described in Chapter 4, except with negligible probability in the computational security parameter λ.

Proof. A server is only blamed if (1) there is a (signed) batch of envelopes showing deviation from the protocol or (2) the server refuses to produce a valid proof of correct decryption. (1) and (2) does not apply to honest servers who follow the protocol. Hence, if an honest server is blamed then a malicious party was capable of forging the honest server's signature, which can only happen with negligible probability. □

5.2 Onion and boomerang encryption

5.2.1 Onion encryption

Remark 3. We require that the onion encryption (and later boomerang encryption) provide Indistinguishability under Chosen-Ciphertext Attacks (IND-CCA security) [11]. While it is conceivable to require weaker security guarantees from the encryption (e.g., Indistinguishability under Chosen-*Plaintext* Attacks; IND-CPA security) our use of these primitives in Trellis requires IND-CCA security for efficient blame assignment.

Let $\mathcal{E} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be any symmetric-key encryption scheme and $\ell \geq 2$. For

convenience, we define:

$$\begin{aligned}
\overrightarrow{\mathsf{onion}}.\mathsf{Enc}(\boldsymbol{sk}, m) &:= \mathsf{Enc}_{\mathsf{sk}_1}(\mathsf{Enc}_{\mathsf{sk}_2}(\ldots \mathsf{Enc}_{\mathsf{sk}_\ell}(m)\ldots)),\\
\overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c) &:= \mathsf{Dec}_{\mathsf{sk}_1}(\mathsf{Dec}_{\mathsf{sk}_2}(\ldots \mathsf{Dec}_{\mathsf{sk}_\ell}(c)\ldots)),\\
\overleftarrow{\mathsf{onion}}.\mathsf{Enc}(\boldsymbol{sk}, m) &:= \mathsf{Enc}_{\mathsf{sk}_\ell}(\mathsf{Enc}_{\mathsf{sk}_{\ell-1}}(\ldots \mathsf{Enc}_{\mathsf{sk}_1}(m)\ldots)),\\
\overleftarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c) &:= \mathsf{Dec}_{\mathsf{sk}_\ell}(\mathsf{Dec}_{\mathsf{sk}_{\ell-1}}(\ldots \mathsf{Dec}_{\mathsf{sk}_1}(c)\ldots)),
\end{aligned}$$

where $\boldsymbol{sk} := (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$ are secret keys sampled i.i.d. according to KeyGen.

Onion encryption must satisfy *correctness* and *IND-CCA security*, defined as follows.

Correctness. For all security parameters λ, integers $\ell \geq 2$, and messages m in the message space,

$$\Pr\left[\begin{array}{l}\boldsymbol{sk} := (\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell) \text{ where } \mathsf{sk}_i \leftarrow \mathcal{E}.\mathsf{KeyGen}(1^\lambda)\ \forall i \in \{1, \ldots, \ell\} : \\ \overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, \overrightarrow{\mathsf{onion}}.\mathsf{Enc}(\boldsymbol{sk}, m)) = \overleftarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, \overleftarrow{\mathsf{onion}}.\mathsf{Enc}(\boldsymbol{sk}, m)) = m\end{array}\right] = 1,$$

where the probability is over the randomness of $\mathcal{E}.\mathsf{KeyGen}$, $\overrightarrow{\mathsf{onion}}.\mathsf{Enc}$, and $\overleftarrow{\mathsf{onion}}.\mathsf{Enc}$.

IND-CCA security. Let $\mathcal{E} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be an IND-CCA secure symmetric-key encryption scheme. For all integers $\ell \geq 2$, there exists a negligible function negl and security parameter λ such that for all PPT adversaries $\mathcal{A}$ it holds that:

$$\Pr\left[\ \textsc{OnionCCA}_{\mathsf{onion},\mathcal{E},\ell,\mathcal{A}}(\lambda) = \mathsf{yes}\ \right] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where $\textsc{OnionCCA}_{\mathsf{onion},\mathcal{E},\ell,\mathcal{A}}(\lambda)$ is defined in Figure 5-1 and $\mathsf{onion} \in \left\{\overrightarrow{\mathsf{onion}}, \overleftarrow{\mathsf{onion}}\right\}$. In words, the adversary can corrupt an arbitrary subset of keys in an attempt to win the CCA game. Additionally, when given the challenge ciphertext c_b, the adversary can continue to query for arbitrary decryptions provided it does not query for any envelopes present in the challenge c_b.

Theorem 2 (Correctness and security of onion encryption [16, 32] (informal))**.** Fix integer $\ell \geq 2$. If $\mathcal{E} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is a correct and IND-CCA-secure symmetric-key encryption scheme, then $\mathsf{onion} \in \left\{\overrightarrow{\mathsf{onion}}, \overleftarrow{\mathsf{onion}}\right\}$ is correct and IND-CCA secure

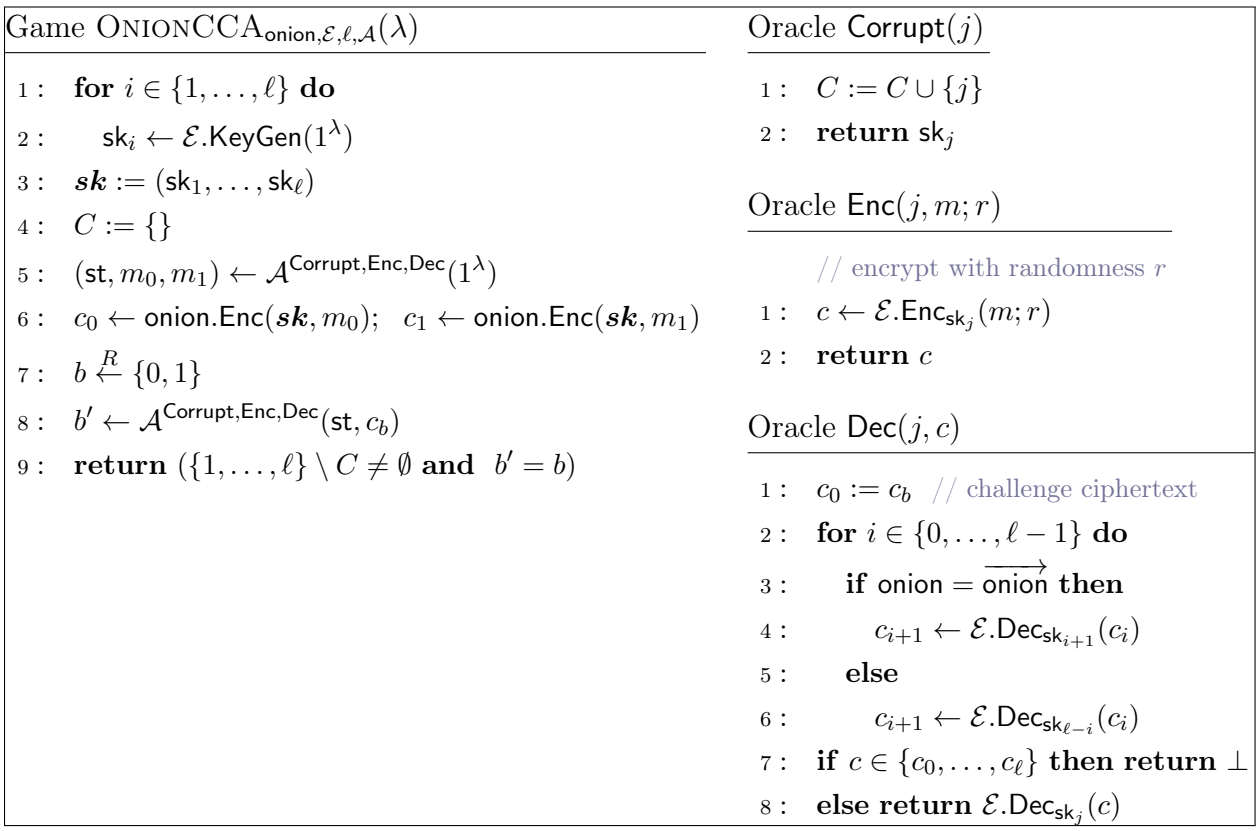

Figure 5-1: Onion encryption IND-CCA security game.

(as defined in Figure 5-1).

Proof (sketch). Correctness follows immediately from the correctness of $\mathcal{E}$. The proof of IND-CCA security for onion encryption is folklore and follows from a straightforward reduction to the IND-CCA security of $\mathcal{E}$. If a PPT $\mathcal{A}$ wins the $\text{ONIONCCA}_{\mathsf{onion},\mathcal{E},\ell,\mathcal{A}}(\lambda)$ game, then there exists a PPT $\mathcal{B}$ that wins the IND-CCA game for the underlying encryption scheme by simulating all but one of the encryption keys and answering queries issued by $\mathcal{A}$ by querying its own IND-CCA oracles for the one key it does not have. $\mathcal{B}$ outputs exactly as $\mathcal{A}$ does. Therefore, if the underlying encryption scheme is IND-CCA secure, then the IND-CCA security of onion encryption follows. □

5.2.2 Boomerang encryption

In this section, we formalize the properties required of boomerang encryption. Let $\overrightarrow{\mathsf{onion}}$ and $\overleftarrow{\mathsf{onion}}$ be as defined in Appendix 5.2.1. For convenience, we define:

$$\mathsf{boomerang.Enc}(\boldsymbol{sk}, \overrightarrow{m}, \overleftarrow{m}) := \overrightarrow{\mathsf{onion}}(\boldsymbol{sk}, \overrightarrow{m} \parallel \overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, \overleftarrow{m})),$$
$$\mathsf{boomerang.Dec}(\boldsymbol{sk}, c) := \overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c)[0] \parallel \overleftarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, \overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c)),$$

where $\overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c)$ is parsed as a tuple $(\overrightarrow{m}, \overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, \overleftarrow{m}))$ so that $\overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c)[0] = \overrightarrow{m}$.

Boomerang encryption must satisfy *correctness* and IND-CCA security, defined as follows.

Correctness. Let $\mathcal{E} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a IND-CCA secure symmetric key encryption scheme. For all security parameters λ, integers $\ell \geq 2$, and pairs of messages $(\overrightarrow{m}, \overleftarrow{m})$ in the message space,

$$\Pr\left[\begin{array}{l} \boldsymbol{sk} := (\mathsf{sk}_1, \ldots, \mathsf{sk}_{2\ell}) \text{ where } \mathsf{sk}_i \leftarrow \mathcal{E}.\mathsf{KeyGen}(1^\lambda)\ \forall i \in \{1, \ldots, 2\ell\} : \\ \mathsf{boomerang.Dec}(\boldsymbol{sk}, \mathsf{boomerang.Enc}(\boldsymbol{sk}, \overrightarrow{m}, \overleftarrow{m})) = \overrightarrow{m} \| \overleftarrow{m} \end{array}\right] = 1,$$

where the probability is over the randomness of KeyGen and $\mathsf{boomerang.Enc}$.

IND-CCA security. Let $\mathcal{E} := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a IND-CCA secure encryption scheme. For all integers $\ell \geq 2$, there exists a negligible function negl and security parameter λ such that for all PPT adversaries $\mathcal{A}$ it holds that:

$$\Pr\Big[\ \textsc{BoomerangCCA}_{\mathsf{boomerang},\mathcal{E},\ell,\mathcal{A}}(\lambda) = \mathsf{yes}\ \Big] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where $\textsc{BoomerangCCA}_{\mathsf{boomerang},\mathcal{E},\ell,\mathcal{A}}(\lambda)$ is defined in Figure 5-2.

Theorem 3 (Security of boomerang encryption)**.** Fix integer $\ell \geq 2$. If $\mathsf{onion} \in \left\{\overrightarrow{\mathsf{onion}}, \overleftarrow{\mathsf{onion}}\right\}$ meets the correctness and IND-CCA security properties of onion en-

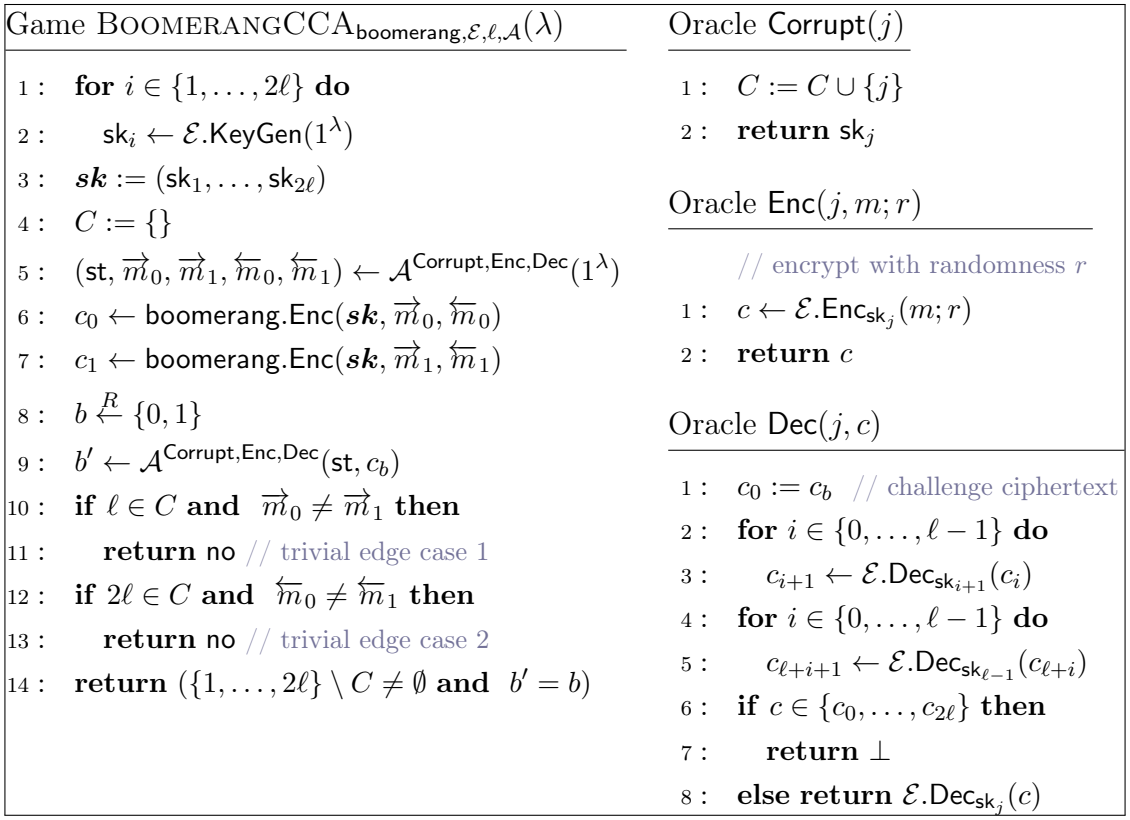

Figure 5-2: Boomerang encryption IND-CCA security game.

cryption, then the boomerang encryption scheme defined as

$$\mathsf{boomerang.Enc}(\boldsymbol{sk}, \overrightarrow{m}, \overleftarrow{m}) := \overrightarrow{\mathsf{onion}}(\boldsymbol{sk}, \overrightarrow{m} \parallel \overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, \overleftarrow{m}))),$$
$$\mathsf{boomerang.Dec}(\boldsymbol{sk}, c) := \overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c)[0] \parallel \overleftarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, \overrightarrow{\mathsf{onion}}.\mathsf{Dec}(\boldsymbol{sk}, c)),$$

is correct and IND-CCA secure (as defined in Figure 5-2).

Proof (sketch). Suppose, towards contradiction, that there exists a PPT $\mathcal{A}$ that wins the IND-CCA security game of boomerang encryption with non-negligible advantage $\delta(\lambda)$. Construct a PPT $\mathcal{B}$ that wins the onion IND-CCA security game with the same advantage. On input (1^λ), $\mathcal{B}$ samples ℓ keys according to KeyGen and either sets these keys as the keys for $\overrightarrow{\mathsf{onion}}$ or $\overleftarrow{\mathsf{onion}}$ (either is without loss of generality). At this point, $\mathsf{boomerang}$ is defined over 2ℓ keys, of which $\mathcal{B}$ only knows ℓ. $\mathcal{B}$ runs $\mathcal{A}$ on (1^λ) and answers all $\mathsf{Corrupt}$, Enc, and Dec queries as follows. If queried on an index for which it knows the secret key sk_j, $\mathcal{B}$ answers the query accordingly; otherwise, it queries the

$\mathsf{Corrupt}$, Enc, or Dec oracle for onion encryption and responds as they do. Upon receiving $(\mathsf{st}, \overrightarrow{m}_0, \overrightarrow{m}_1, \overleftarrow{m}_0, \overleftarrow{m}_1)$, $\mathcal{B}$ outputs $(\mathsf{st}, \overrightarrow{m}_0 \parallel \overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, \overleftarrow{m}_0), \overrightarrow{m}_1 \parallel \overleftarrow{\mathsf{onion}}(\boldsymbol{sk}, \overleftarrow{m}_1))$ (by querying the Enc oracle as needed to generate the onion encryption). Upon receiving challenge (st, c_b), $\mathcal{B}$ runs $\mathcal{A}$ on input (st, c_b). $\mathcal{B}$ answers Enc and Dec queries as before but outputs $\perp$ when queried on any challenge ciphertext ($\mathcal{B}$ can check the necessary condition by using the secret keys it knows and querying the Dec oracle). Finally, $\mathcal{B}$ and outputs as $\mathcal{A}$ does. It is easy to check that $\mathcal{B}$ succeeds whenever $\mathcal{A}$ succeeds, contradicting the assumption that $\mathsf{onion} \in \left\{\overrightarrow{\mathsf{onion}}, \overleftarrow{\mathsf{onion}}\right\}$ meets the IND-CCA security property of onion encryption. □

5.2.3 Boomerang routing and proof-of-delivery

In Trellis, boomerang encryption is used to route a messages through a chain of ℓ servers (and back). Here, we formalize the key property of boomerang routing that we used in Trellis: proof of delivery.

Lemma 1 (Boomerang decryption requires all keys)**.** *If the* $\mathsf{boomerang}$ *scheme satisfies the correctness and IND-CCA security definitions of boomerang encryption, then a successful boomerang decryption of a ciphertext implies that each of the* 2ℓ *secret keys were used to decrypt the boomerang ciphertext.*

Proof. If a secret key was *not* used in the decryption process then it is possible to construct an adversary $\mathcal{A}$ that can decrypt a layer in the nested ciphertext *without* the secret key for that layer. This immediately gives rise to an adversary that wins the $\textsc{BoomerangCCA}_{\mathsf{boomerang},\mathcal{E},\ell,\mathcal{A}}(\lambda)$ game (Figure 5-2), contradicting the IND-CCA security of the boomerang encryption scheme. □

Claim 9 (Boomerang decryption as proof-of-delivery)**.** Let $r \in \{0,1\}^\kappa$ be a random secret nonce known only to the encryptor where κ is a statistical security parameter. A successful boomerang decryption consisting of $\overleftarrow{m} = r$ given a boomerang encryption of $(\overrightarrow{m} \in \{0,1\}^*,\ \overleftarrow{m} = r)$ implies that $\overrightarrow{m}$ was decrypted using secret key sk_ℓ. In boomerang routing, this implies that the ℓth server decrypted (and thus received) $\overrightarrow{m}$, with probability at least $1 - 2^{-\kappa}$.

Proof. Suppose the claim is false. Then, either (1) the secret key sk_ℓ was not used which contradicts Lemma 1 or (2) a *different* (potentially shorter) boomerang encryption of r was generated, without using sk_ℓ. However, because r is random and kept secret by the decryptor, this can only happen with probability $2^{-\kappa}$. □

5.3 Anonymous routing tokens

5.3.1 Properties

We present the syntax of ARTs in Definition 1. Here, we present the formal properties we require from ARTs.

Notation. We define the Setup algorithm to output a $\mathsf{crs} := (\mathbb{Z}_p, \mathbb{G}, g)$ where $\mathbb{G}$ is a group of prime order p with generator g; we assume that the prime p is proportional to the security parameter λ. We let $s \geq 1$ be the number of signers holding partial token signing keys. Finally, for a fixed integer $N \geq 2$ we let $\mathbb{S}$ be the set of N server identifiers.

Modeling the signing protocol. To simplify our analysis, we will model the signing protocol as a triple of (non-interactive) algorithms. This model captures one-round protocols instantiated between the user and the signers and corresponds to the three steps in the construction described in Section 3.4.1. For convenience, we let $\mathcal{P}$ be a distribution over $(S_\ell, A_\ell, \mathsf{vk}_\ell)$ where S_ℓ is the ℓth server identity, A_ℓ is a random Diffie-Hellman message, and vk_ℓ is a signature verification key. We denote by $\mathsf{payload}$ a random sample from $\mathcal{P}$.

- $\mathsf{User}_0(\mathsf{tvk}, \mathsf{payload}_\ell) \to (\mathsf{st}_\ell, T_\ell)$: takes as input a token payload $\mathsf{payload}_\ell$; outputs secret user state st and blind token T_ℓ.
- $\mathsf{PartSign}(\mathsf{tvk}, x_i, T_\ell) \to W_{\ell,i}$: takes as input a partial signing key x_i and blind token T_ℓ; outputs a partial signature $W_{\ell,i}$.
- $\mathsf{User}_1(\mathsf{st}_\ell, W_{\ell,1}, \ldots, W_{\ell,s}) \to \mathsf{t}_\ell$: takes as input the secret user state st_ℓ and s partial signatures; outputs signed token t_ℓ.

Completeness. An honest user engaging with honest signers obtains a valid token which makes Verify output yes. Formally, for all security parameters λ and all payloads $\mathsf{payload}_\ell$, it holds that:

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda); \\ (\mathsf{tvk}, x_1, \ldots, x_s) \leftarrow \mathsf{KeyGen}(\mathsf{crs}); \\ (\mathsf{st}_\ell, T_\ell) \leftarrow \mathsf{User}_0(\mathsf{tvk}, \mathsf{payload}_\ell); \\ W_{\ell,i} \leftarrow \mathsf{PartSign}(\mathsf{tvk}, x_i, T_\ell)\ \forall i \in \{1, \ldots, s\}; \\ \mathsf{t}_\ell \leftarrow \mathsf{User}_1(\mathsf{st}_\ell, W_{\ell,1}, \ldots, W_{\ell,s}); \\ S_{\ell+1} \leftarrow \mathsf{NextServer}(\mathsf{tvk}, \mathsf{t}_\ell) \end{array} : \mathsf{Verify}(\mathsf{tvk}, S_{\ell+1}, \mathsf{payload}_\ell, \mathsf{t}_\ell) = \mathsf{yes} \right] = 1,$$

where the probability is over the randomness of Setup, KeyGen, User_0, $\mathsf{PartSign}$, and User_1.

Unforgeability. No (strict) subset of signers and/or user should be able to generate valid tokens that are accepted by Verify, even after seeing $\ell - 1$ valid tokens. Formally, an ART scheme is said to be *one-more unforgeable* if there exists a negligible function negl such that for all PPT adversaries $\mathcal{A}$, and any $\ell \geq 1$:

$$\Pr\Big[\ \textsc{Omu}_{\mathsf{ART},\mathcal{A},\ell}(\lambda) = \mathsf{yes}\ \Big] \leq \mathsf{negl}(\lambda),$$

where $\textsc{Omu}_{\mathsf{ART},\mathcal{A},\ell}(\lambda)$ is defined Figure 5-3. In words, the adversary's task is to forge a fresh token after seeing $\ell - 1$ signed tokens of its choosing, without corrupting all the signing keys.

Unlinkability. Each token produced through the Sign protocol should be unlinkable to the inputs given to the signers (even if all signers collude, which we model as the adversary choosing secret keys after being given the crs). Formally, an ART scheme is said to be unlinkable if there exists a negligible function negl such that for all PPT adversaries $\mathcal{A}$, and any $\ell \geq 2$:

$$\Pr\Big[\ \textsc{Unlink}_{\mathsf{ART},\mathcal{P},\mathcal{A},\ell(\lambda)} = \mathsf{yes}\ \Big] \leq \frac{1}{\ell} + \mathsf{negl}(\lambda),$$

Game $\textsc{Omu}_{\mathsf{ART},\mathcal{A},\ell}(\lambda)$

$\mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda)$
$(\mathsf{tvk}, x_1, \ldots, x_s) \leftarrow \mathsf{ART.KeyGen}(\mathsf{crs})$
$q_1, \ldots, q_s := 0; \quad C := \{\}$
$\{(\_, \mathsf{payload}_i, \mathsf{t}_i) \mid 1 \le i \le \ell\} \leftarrow \mathcal{A}^{\mathsf{Corrupt},\mathsf{PartSign},\mathsf{Verify}}(\mathsf{crs}, \mathsf{tvk})$
for $i \in \{1, \ldots, \ell\}$ **do**
  $S_{i+1} \leftarrow \mathsf{ART.NextServer}(\mathsf{t}_i)$
$\tilde{C} := \{1, \ldots, s\} \setminus C$
return ($\tilde{C} \neq \emptyset$ **and** $\forall i \in \tilde{C} \;\; q_i \le \ell - 1$ **and**
  $\forall i \neq j \in \{1, \ldots, \ell\} \;\; \mathsf{t}_i \neq \mathsf{t}_j$ **and**
  $\forall i \in \{1, \ldots, \ell\} \;\; \mathsf{ART.Verify}(\mathsf{tvk}, S_{i+1}, \mathsf{payload}_i, \mathsf{t}_i) = \mathsf{yes}$)

Oracle $\mathsf{Corrupt}(j)$

$C := C \cup \{j\}$
return x_j

Oracle $\mathsf{PartSign}(j, T_i)$

$q_j := q_j + 1$
$W_{i,j} \leftarrow \mathsf{ART.PartSign}(\mathsf{tvk}, x_j, T_i)$
return $W_{i,j}$

Oracle $\mathsf{Verify}(S_{j+1}, \mathsf{payload}_j, \mathsf{t}_j)$

return $\mathsf{ART.Verify}(\mathsf{tvk}, S_{j+1}, \mathsf{payload}_j, \mathsf{t}_j)$

Figure 5-3: One-more unforgeability game for anonymous routing tokens (ARTs).

where $\textsc{Unlink}_{\mathsf{ART},\mathcal{P},\mathcal{A},\ell(\lambda)}$ is defined in Figure 5-4.

Unpredictability. The output of NextServer should be unpredictable given only the payload $\mathsf{payload}_\ell$. Formally, the ART scheme is said to generate unpredictable links

Game $\textsc{Unlink}_{\mathsf{ART},\mathcal{P},\mathcal{A},\ell(\lambda)}$

1 : $\mathsf{crs} \leftarrow \mathsf{ART.Setup}(1^\lambda)$
2 : $(\mathsf{st}, \mathsf{tvk}, x_1, \ldots, x_s) \leftarrow \mathcal{A}(\mathsf{crs})$
3 : $q_0 := 0; \quad q_1 := 0; \quad Q := \{\}$
4 : $(\mathsf{st}, T_1, \ldots, T_\ell) \leftarrow \mathcal{A}^{\mathsf{User}_0, \mathsf{User}_1}(\mathsf{st})$
5 : **if** $Q = \emptyset$ **then return** no
6 : **for** $i \in \{1, \ldots, \ell\}$ **do**
7 : **for** $j \in \{1, \ldots, s\}$ **do**
8 : $W_{i,j} \leftarrow \mathsf{ART.PartSign}(x_j, T_j)$
9 : $\mathsf{t}_i \leftarrow \mathsf{ART.User}_1(\hat{\mathsf{st}}_i, W_{i,1}, \ldots, W_{i,s})$
10 : $j \xleftarrow{R} Q$
11 : $j' \leftarrow \mathcal{A}(\mathsf{st}, (\mathsf{payload}_j, \mathsf{t}_j))$
12 : **return** $j' = j$

Oracle $\mathsf{User}_0()$

1 : $q_0 := q_0 + 1$
2 : $\mathsf{payload}_{q_0} \xleftarrow{R} \mathcal{P}$
3 : $(\hat{\mathsf{st}}_{q_0}, T_{q_0}) \leftarrow \mathsf{ART.User}_0(\mathsf{tvk}, \mathsf{payload}_{q_0})$
4 : $Q := Q \cup \{q_0\}$
5 : **return** T_{q_0}

Oracle $\mathsf{User}_1(j, W_1, \ldots, W_s)$

1 : **if** $j \notin Q$ **then return** $\perp$
2 : $\mathsf{t}_j \leftarrow \mathsf{ART.User}_1(\hat{\mathsf{st}}_j, W_1, \ldots, W_s)$
3 : $S_{j+1} \leftarrow \mathsf{ART.NextServer}(\mathsf{t}_j)$
4 : **if** $\mathsf{Verify}(\mathsf{tvk}_j, S_{j+1}, \mathsf{payload}_j, \mathsf{t}_j) = \mathsf{yes}$ **then**
5 : $Q := Q \setminus \{j\}$
6 : $q_1 := q_1 + 1$
7 : **return** $(\mathsf{payload}_j, \mathsf{t}_j)$

Figure 5-4: Unlinkability game for the anonymous routing tokens (ARTs).

if there exists a negligible function negl such that for all $S_\ell \in \mathbb{S}$ and all $S^* \in \mathbb{S}$,

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathsf{Setup}(1^\lambda); \\ (\mathsf{tvk}, x_1, \ldots, x_s) \leftarrow \mathsf{KeyGen}(\mathsf{crs}); \\ (\mathsf{st}_\ell, T_\ell) \leftarrow \mathsf{User}_0(\mathsf{tvk}, S_\ell); \\ W_{\ell,i} \leftarrow \mathsf{PartSign}(\mathsf{tvk}, x_i, T_\ell)\ \forall i \in \{1, \ldots, s\}; \\ \mathsf{t}_\ell \leftarrow \mathsf{User}_1(\mathsf{st}_\ell, W_{\ell,1}, \ldots, W_{\ell,s}); \\ S_{\ell+1} \leftarrow \mathsf{NextServer}(\mathsf{tvk}, \mathsf{t}_\ell) \end{array} : S_{\ell+1} = S^* \right] = \frac{1}{|\mathbb{S}|} - \mathsf{negl}(\lambda),$$

where the probability is over the randomness of Setup, KeyGen, User_0, $\mathsf{PartSign}$, and User_1.

5.3.2 Security analysis

Chosen-target Gap Diffie-Hellman (CTGDH). We prove the *unforgeability* property of our ART tokens under the CTGDH assumption, following prior work on anonymous tokens [58]. Informally, the assumption is related to computational Diffie-

Hellman (CDH) assumption in *gap* groups where *decisional* Diffie-Hellman (DDH) is easy but the CDH assumption is assumed to be computationally intractable. Specifically, certain bilinear groups (such as the ones we use to construct ARTs) have an efficiently computable pairing, which makes the decisional variant of Diffie-Hellman easy [12]. In these groups, the gap Diffie-Hellman (GDH) assumption is the bilinear counterpart to the classic CDH assumption. The chosen-target variant of GDH, is introduced by Kreuter et al. [58] to capture a setting where the adversary can compute $\ell - 1$ instances of CDH with the help of an oracle *and* an oracle for DDH. The adversary is tasked with generating ℓ CDH instances (one more than it queried for, hence *one-more unforgeability*). This assumption is rather natural (see Kreuter et al. [58, Section 2.1 & Appendix A.2] for details and how it relates to prior work) and makes our analysis straightforward. We describe the CTGDH security game in Figure 5-5 and refer the reader to Kreuter et al. [58] for additional information.

Assumption 1 (CTGDH [58])**.** Let Setup be a group generator algorithm that, given a security parameter 1^λ, outputs a group $\mathbb{G}$ of prime order p where $\lambda = \lceil \log_2 p \rceil$ and a nothing-up-my-sleeve generator[1] g of $\mathbb{G}$. CTGDH is said to hold if there exists a security parameter λ and negligible function negl such that for all PPT $\mathcal{A}$ and any $\ell \geq 1$:

$$\Pr\Big[\ \mathrm{CTGDH}_{\mathsf{Setup},\mathcal{A},\ell}(\lambda) = \mathsf{yes}\ \Big] \leq \mathsf{negl}(\lambda),$$

where $\mathrm{CTGDH}_{\mathsf{Setup},\mathcal{A},\ell}(\lambda)$ is defined in Figure 5-5.

Theorem 4 (ART Completeness)**.** The ART construction presented in Section 3.4.1 satisfies the completeness property of Appendix 5.3.1.

Proof. Consider any $S_\ell \in \mathbb{S}$. The output of User_0 is $T_\ell := \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^{1/r}$ where r is random in $\mathbb{Z}_p$. The output of $\mathsf{PartSign}$ is then $W_{\ell,i} := T_\ell^{x_i}$ and so it follows that $\mathsf{t}_\ell = \prod_{i=1}^{s} W_{\ell,i} = \prod_{i=1}^{s} T_i^{x_i} = (\prod_{i=1}^{s} T_i)^x = \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^{x/r}$. Given this, the output of User_1 is t_ℓ. By the above, we have that $\mathsf{t}_\ell = \big(\mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^{x/r}\big)^r = \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^x$.

[1] In practice, this simply means that Setup must be computed by a trusted setup process to avoid having any party unfairly choosing the generator.

Game $\text{CTGDH}_{\mathsf{Setup},\mathcal{A},\ell}(\lambda)$

1 : $(\mathbb{G}, p, g) \leftarrow \mathsf{Setup}(1^\lambda)$
2 : $x \xleftarrow{R} \mathbb{Z}_p$; $\quad \mathsf{pk} \leftarrow g^x$
3 : $q := 0$; $\quad Q := []$
4 : $\{(w_i, z_i) \mid 1 \le i \le \ell\} \leftarrow \mathcal{A}^{\mathsf{Target},\mathsf{Help},\mathsf{DDH}}((\mathbb{G}, p, g), \mathsf{pk})$
5 : **for** $i \in \{1, \ldots, \ell\}$ **do**
6 : $\quad$ **if** $w_i \notin Q$ **then return** no
7 : $\quad y_i := Q[w_i]$
8 : **return** $(q \le \ell - 1$ **and**
9 : $\qquad \forall i \ne j \in \{1, \ldots, \ell\} \;\; w_i \ne w_j$ **and**
10 : $\qquad \forall i \in \{1, \ldots, \ell\} \;\; (y_i)^x = z_i)$

Oracle $\mathsf{Target}(w)$

1 : **if** $w \in Q$ **then**
2 : $\quad y := Q[w]$
3 : **else**
4 : $\quad y \xleftarrow{R} \mathbb{G}$
5 : $\quad Q[w] := y$
6 : **return** y

Oracle $\mathsf{Help}(y)$

1 : $q := q + 1$
2 : **return** y^x

Oracle $\mathsf{DDH}(y, z)$

1 : **return** $y = z^x$

Figure 5-5: The Chosen-target gap Diffie–Hellman game [58].

It follows that $e(\mathsf{t}_\ell, g) = \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^x \in \mathbb{G}_{\mathrm{T}}$ and $e(y_\ell, g^x) = \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^x \in \mathbb{G}_{\mathrm{T}}$, only when $y_\ell := \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})$. As such, we have that the following two conditions are satisfied:

(1) $e(\mathsf{t}_\ell, g) = e(y_\ell, g^x)$ and

(2) $S'_{\ell+1} = \mathsf{Hash}_{\mathbb{Z}_p}(\mathsf{t}) = S_{\ell+1}$.

Together, (1) and (2) imply that Verify outputs yes, as required. □

Theorem 5 (ART Unforgeability)**.** The ART construction presented in Section 3.4.1 satisfies the unforgeability property of Appendix 5.3.1 under the CTGDH assumption.

Proof. Suppose that $\mathcal{A}$ wins the $\text{OMU}_{\mathsf{ART},\mathcal{A},\ell}(\lambda)$ game with probability $\delta(\lambda)$, for some non-negligible function δ. Construct an adversary $\mathcal{B}$ that wins the $\text{CTGDH}_{\mathsf{Setup},\mathcal{A},\ell}(\lambda)$ with probability $\frac{\delta(\lambda)}{s}$, where s is the number of signers. $\mathcal{B}$ works as follows.

1. On input $(\mathbb{Z}_p, \mathbb{G}, g)$ and $\mathsf{pk} := g^x$, set $\mathsf{crs} := (\mathbb{Z}_p, \mathbb{G}, g)$.
2. Sample $j^* \xleftarrow{R} \{1, \ldots, s\}$, $x_1, \ldots, x_s \xleftarrow{R} \mathbb{Z}_p$ and set $\mathsf{pk}' = g^x \cdot \prod_{i=1, i \ne j^*}^{s} g^{x_i}$.
3. Run $\mathcal{A}(\mathsf{crs}, \mathsf{pk}')$ and answer $\mathsf{Hash}_{\mathbb{G}}$, Corrupt, PartSign, and Verify queries as follows.
   - For each query to $\mathsf{Hash}_{\mathbb{G}}$, query the Target oracle and respond as it does.

- For each $\mathsf{Corrupt}$ query of the form j:
  - If $j = j^*$ then **abort**,
  - Else respond with x_j.
- For each $\mathsf{PartSign}$ query of the form (j, T_i):
  - If $j = j^*$ then query the Help oracle on input T_i to get $W_{i,j^*} := (T_i)^x$,
  - Else set $W_{i,j} := (T_i)^{x_j}$.
  - Respond with $W_{i,j}$.
- For each Verify query of the form $(S_{j+1}, \mathsf{payload}_j, \mathsf{t}_j)$:
  - Query the Target oracle on input $\mathsf{payload}_j$ to get y.
  - Query the DDH oracle on input (t_j, g) and (y, g^x) to get answers a_0 and a_1, respectively.
  - If $a_0 \neq a_1$ return no,
  - Else query the random oracle on input t_j to get S'_{j+1}.
  - If $S'_{j+1} \neq S_{j+1}$ return no.
  - Else return yes.

4. Obtain from $\mathcal{A}$ the output $(\_, \mathsf{payload}_i, \mathsf{t}_i)$ for $i \in \{1, \ldots, \ell\}$.
5. Set $w_i := \mathsf{payload}_i$ for $i \in \{1, \ldots, \ell\}$.
6. Compute $\Delta := \sum_{i=1, i \neq j^*}^{s} x_i$.
7. Set $z_i := \mathsf{t}_i g^{-\Delta}$.
8. Output $\{(w_i, z_i) \mid 1 \leq i \leq \ell\}$.

We now analyze the reduction and argue why $\mathcal{B}$ succeeds with probability $\frac{\delta(\lambda)}{s}$. First, observe that pk' is a uniformly random element of $\mathbb{G}$ and hence matches the distribution expected by $\mathcal{A}$. Similarly, each partial signing key, x_i is a uniformly random element of $\mathbb{Z}_p$ and hence also distributed identically. The responses to the $\mathsf{PartSign}$ and Verify queries are distributed identically to the distribution expected by $\mathcal{A}$ in the $\textsc{Omu}_{\mathsf{ART},\mathcal{A},\ell}(\lambda)$ game. As such, if $\mathcal{A}$ wins the $\textsc{Omu}_{\mathsf{ART},\mathcal{A},\ell}(\lambda)$ game, it must output ℓ tuples of the form $(\_, \mathsf{payload}_i, \mathsf{t}_i)$ such that:

1. the total queries to $\mathsf{PartSign}$ for $j = j^*$ was at most $\ell - 1$,
2. the total unique queries to $\mathsf{Corrupt}$ was at most $s - 1$,
3. $\forall i \neq j \;\; \mathsf{t}_i \neq \mathsf{t}_j$, and
4. $\forall i \;\; \mathsf{ART.Verify}(\mathsf{tvk}, S_{j+1}, \mathsf{payload}_i, \mathsf{t}_i) = \mathsf{yes}$.

Because the total number of queries to $\mathsf{PartSign}$ for $j = j^*$ was at most $\ell - 1$, $\mathcal{B}$ queries the Help oracle at most $\ell - 1$ times. Because the total unique queries to $\mathsf{Corrupt}$ was at most $s - 1$, the probability that $\mathcal{A}$ doesn't query j^* is at least $\frac{1}{s}$. By the last two properties (and the construction of $\mathsf{ART.Verify}$), it holds that each t_i is unique and, moreover, $\mathsf{t}_i = \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload}_i)^x$. It follows that $z_i = (y_i)^x$, where $y_i = Q[w_i] = Q[\mathsf{payload}]$ is as defined in Figure 5-5. In sum, we get that $\forall i \neq j \;\; w_i \neq t_j$ and $\forall i \;\; (y_i)^x = z_i$. These conditions make $\mathcal{B}$ win with the same probability as $\mathcal{A}$, *provided that $\mathcal{B}$ doesn't abort.* The probability that $\mathcal{B}$ aborts is entirely conditioned on $\mathcal{A}$ querying $\mathsf{Corrupt}$ on index j^* (for which $\mathcal{B}$ doesn't have the secret key). Because j^* is chosen at random, the probability that $\mathcal{B}$ aborts is at most $\frac{1}{s}$ ($\mathcal{A}$ queries at most $s - 1$ keys). As such, the probability that $\mathcal{B}$ wins the $\textsc{ctgdh}_{\mathsf{Setup},\mathcal{A},\ell}(\lambda)$ game is $\frac{\delta(\lambda)}{s}$, concluding the proof.

□

Theorem 6 (ART Unlinkability)**.** The ART construction presented in Section 3.4.1 unconditionally satisfies the unlinkability property described above.

Proof. Each blind token T_i output by User_0 is of the form $\mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^{\frac{1}{r}}$. Because $r \in \mathbb{Z}_p$ is random and $\mathsf{Hash}_{\mathbb{G}}$ is a random oracle, it follows that T_i is uniformly random in $\mathbb{G}$. The output of User_1 is t_i where $\mathsf{t}_i = ((T_i)^x)^r = ((T_i)^r)^x = \mathsf{Hash}_{\mathbb{G}}(\mathsf{payload})^x$ (recall that the user verifies that the token is computed correctly with respect to tvk using Verify, which prevents "tagging" the token with a maliciously computed $x^* \neq x$ [28]). As such, t_i is independent of T_i and well-formed if the user accepts. Finally, the payload $\mathsf{payload}_i$ is chosen independently of r and hence is also independent of T_i. This means that the outputs of User_0 and User_1 are also independent of each other. It then follows that because $\mathcal{A}$ does not query User_1 on at least one index j in $\textsc{Unlink}_{\mathsf{ART},\mathcal{P},\mathcal{A},\ell(\lambda)}$ (otherwise $Q = \emptyset$) and because $(\mathsf{payload}_j, \mathsf{t}_j)$ is only revealed by

the User_1 oracle (which we know $\mathcal{A}$ did not query for the chosen challenge), $\mathcal{A}$ has no advantage in correctly guessing j. Thus, unlinkability of tokens holds unconditionally. □

Theorem 7 (ART Unpredictability)**.** If $\mathsf{Hash}_{\mathbb{G}}$ and $\mathsf{Hash}_{\mathbb{Z}_N}$ are hash functions modeled as random oracles, then the ART construction presented in Section 3.4.1 satisfies the unpredictability property described above.

Proof. Each signature t_i is uniformly random due to $\mathsf{Hash}_{\mathbb{G}}$ and unpredictable due to the signing process (without knowledge of x it is not possible to generate t_i by the one-more unforgeability property). In turn, $S_{i+1} := \mathsf{Hash}_{\mathbb{Z}_N}(\mathsf{t}_i)$ is uniformly random over $\mathbb{Z}_N$ and cannot be predicted by any subset of colluding signers or the user. □

Chapter 6

Implementation and Evaluation

We implement Trellis in Go 1.17. Our implementation is open source [1] and consists of approximately 11,000 lines of code. We implement distributed key generation [55] and proactive secret-sharing [30] using the Kyber library [29]. We use the BLS12-381 [12, 114] elliptic curve to implement ARTs. Symmetric-key encryption is instantiated using AES [23, 81] and digital signatures using EdDSA [52]. Diffie-Hellman key agreement is performed over the ed25519 elliptic curve to match EdDSA. For our empirical comparison to Atom, we use their open source implementation [59].

Environment. We distribute the servers evenly among four geographic regions: Oregon, Virginia, Frankfurt, and Stockholm, to roughly match the distribution of Tor servers [72]. We use the general purpose `m5.xlarge` instances (quad-core; 16 GiB of RAM). We measured median round trip ping times of around 150 ms, with rare latency spikes of up to one second.

Due to high international networking costs, we run most of our evaluations on a simulated network environment informed from a real network deployment. In Figure 6-1, we report the latency of Trellis (broadcast round) under a 100 Mbps and 200 Mbps real network and compare latency to our simulated network deployment. We ensure that the simulated network accurately approximates the real network by evaluating the broadcasting rounds (on pre-established paths). We observe modest differences in latency (less than 20%) between the real and simulated networks.

We use the Linux NetEm [47] tool to model latency and packet loss rates for each geographic region. We apply a Pareto distribution to simulate the infrequent, but large, latency spikes observed in the real network. Properly simulating these latency spikes is important because the all-to-all nature of the mixnet in Trellis is sensitive to the *highest* latency observed across all links. As such, tail latencies can significantly impact performance and lead to high variance in broadcast latency.

Parameters. We set $\epsilon := 2^{-64}$ for the total variational distance of the mixing (see Section 3.9). (We also set the size of each anytrust group so that the probability that *any* anytrust group has all adversaries is at most 2^{-64}.) Note that both of these parameters set the *statistical* security of Trellis. To simplify our evaluation, we fix the message size to 10 kB (over 60× larger compared to the 160 B messages evaluated in Atom [62]). We vary the number of users M, number of servers N, and the fraction of malicious servers f (which influences the number of layers L and anytrust group size s).

6.1 Implementation optimizations

In this section, we briefly highlight some practical optimizations we can exploit to improve concrete performance on Trellis in a deployed environment (we use these straightforward optimizations in our implementation).

- **Reduced authentication overhead:** Since each AES ciphertext is signed, we do not need to include MACs to verify integrity.
- **Synchronizing rounds:** After each server receives and checks the signed message from the other servers for layer ℓ, it can start processing layer $\ell + 1$, provided it avoids the computational side channel of revealing the time to process the last received message. With this optimization, the servers do not need to wait for the worst-case latency, and each server can adapt to the current network conditions on-the-fly.
- **Latency ordering:** Signing and then sending the messages along the longest

latency connections first enables the signing and verification computation latency to occur in parallel with the networking latency.

- **Path establishment receipt latency:** The key feature of the boomerang receipt is that it must pass through an honest server to detect that the message was dropped. However, we can compute the number of servers it must pass through to have passed through one honest server with cryptographically high probability (e.g., with probability greater than $1 - 2^{-64}$) in the same manner as we do for computing the anytrust group size. Then, the backward onion can be truncated at $L' < L$ layers.
- **Message overhead:** The key corresponding to each envelope can be appended by the server from the stored data instead of being included as part of the ciphertext. This reduces the overhead added for each layer. The first non-token containing an envelope along a link should include the key within the ciphertext, but the key can be omitted for later envelopes on that link (so envelopes only contain the next onion ciphertext). The block public signature signs the keys so servers can prove which ones they were told to decrypt with. During traceback, a server produces and proves correct decryption of the first envelope to show a key decrypts into another and is the correct key to send with later envelopes.
- **Path establishment forwarding latency:** We need to wait for the receipt to be processed before we extend the path, but we do not need to wait for the first part of the path that has already been established. Therefore, the forward sending can take place beforehand, in parallel. We can then further reduce overhead by combining the forward envelopes with the same key together.
- **Empirical dummy calculation:** The overhead of dummy envelopes is significant, especially when setting the statistical security to 2^{-64}. To reduce this overhead, we first bound the probability that more envelopes are sent on any link than the dummy cover traffic allows for to 2^{-8}. This ensures that the probability that any 8 links are observable to the network adversary is less than 2^{-64}. To compensate, we increase the number of layers to $L + 8$, in order to have L layers

with unobservable mixing guarantees, as required by Section 3.9. We calculate a theoretical bound by modeling each link as a binomial distribution and bound accordingly [6]. In practice, we observe that the overhead of dummy messages is typically between 100% and 200% and decreases as M increases, see Figure 6-5.

6.2 Evaluation

We evaluate our implementation of Trellis in a simulated network deployment. The goal of our evaluation is to:

- determine the processing overheads of Trellis on the user and on the servers through microbenchmarks,
- measure the network overhead as a function of the number of users and fraction of malicious servers in the mix-net,
- evaluate the overhead of dummy envelopes between layers,
- evaluate how Trellis scales with additional servers, and
- compare Trellis to Atom—the state-of-the-art mix-net based approach for metadata-private anonymous broadcast.

Computational and network overheads. The computational overhead of the broadcasting round is minimal given the relatively lightweight cryptography required: AES-decryption of envelopes (on-par with TLS [92]) and digital signature verification. All the computationally expensive cryptographic operations (e.g., pairings and token generation) are precomputed during path establishment. These computational workloads scale with the number of servers, as we show in Figure 6-4. However, the network latency incurred by routing through L layers does not scale with the number of servers and thus the scalability (for a given number of users) eventually plateaus. Furthermore as N^2 approaches M, the overhead of dummy envelopes increases, as we must always send at least one dummy envelope between each pair of servers. This results in a limit on the horizontal scalability of Trellis. To illustrate this point further, in Figure 6-5, we compare a broadcast round in Trellis on a real network

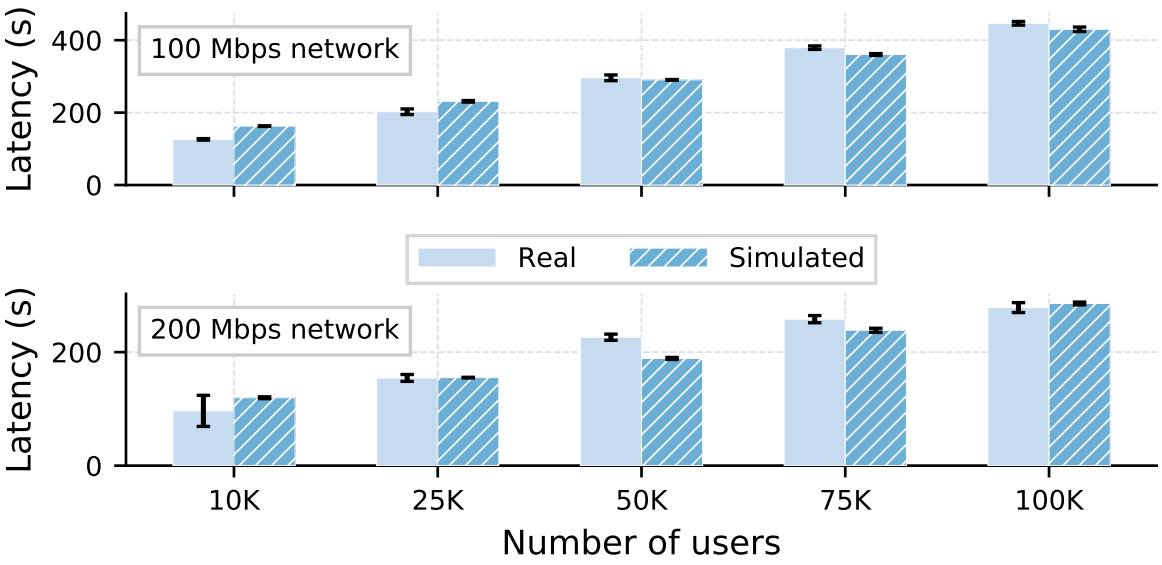

Figure 6-1: Comparison between broadcasting round latency on an international (Oregon, Virginia, Frankfurt, and Stockholm) network and our simulated network. **Top**: 100 Mbps bandwidth connections. **Bottom**: 200 Mbps bandwidth connections.

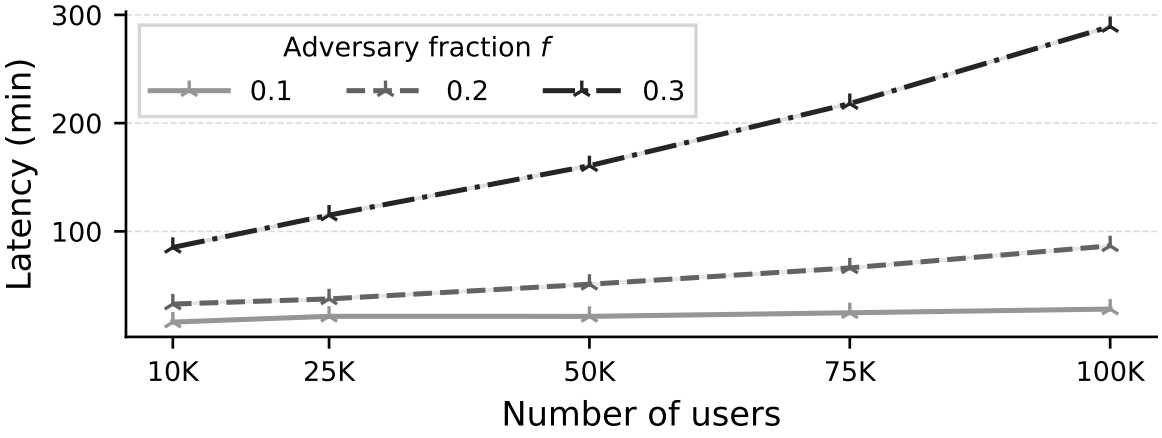

Figure 6-2: Path establishment with different fractions of malicious servers (f). Evaluated on a simulated 200 Mbps network configuration with $N = 128$ servers. (The 95% confidence interval is invisible.)

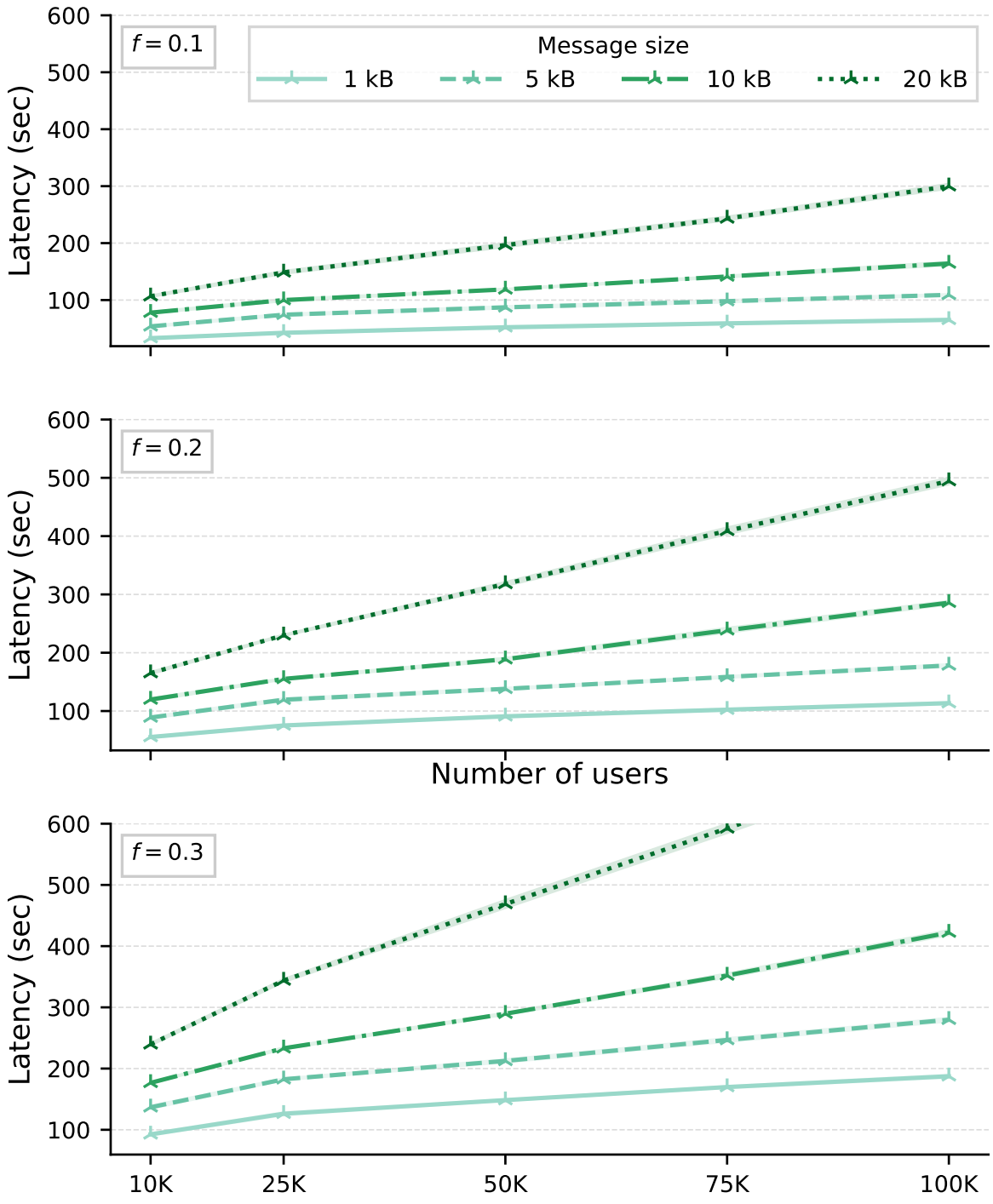

Figure 6-3: Broadcast round scaling with number of users M (messages), message sizes, and different adversary fractions. Evaluated on a simulated 200 Mbps network configuration with $N = 128$ servers. Shaded regions represent a 95% confidence interval (mostly invisible)

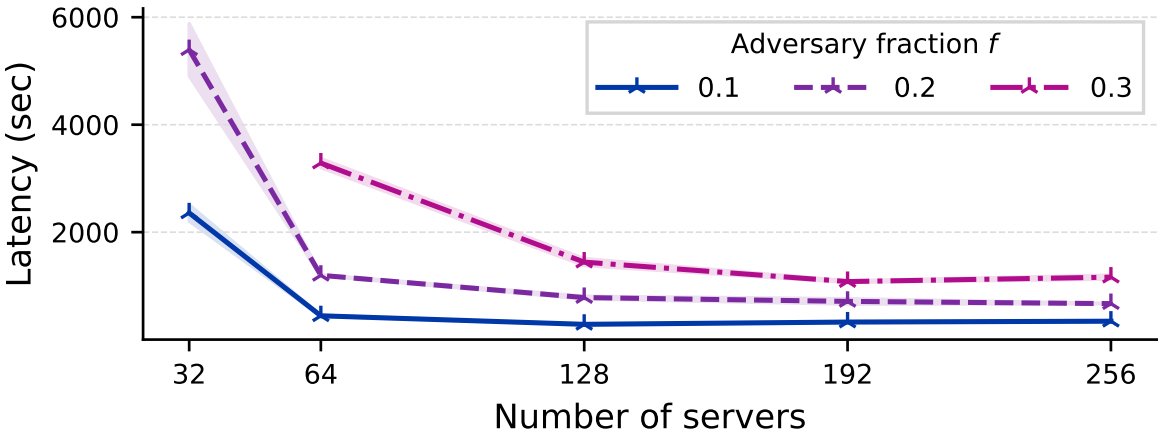

Figure 6-4: Latency decreases with the number of servers added to the network (until reaching a constant overhead) with 200 Mbps bandwidth cap. Number of messages is $M = 2,000,000$ and message size is set to 1 kB. Shaded regions represent a 95% confidence interval.

deployment with and without dummy envelopes included between layers. Dummy envelopes account for roughly 2.5× in latency overhead, see Figure 6-5.

Path establishment. Path establishment can be seen as running consecutive broadcasting rounds with $\ell = 1, 2, \ldots, L$ layers in each consecutive round. Therefore, the time for path establishment is $O(L^2)$, and the time it takes follows accordingly (see Figure 6-3). Path establishment can take one to five hours, depending on parameters. We note that path establishment depends only on the number of users, and not on the message size or number of subsequent broadcast rounds. We report micro-benchmarks for ARTs below:

- ART.PartSign: 0.08 ms (per token). // Overhead on each signer.
- ART.Verify: 1.09 ms (per token). // Token signature verification.

Broadcasting rounds. Once the path establishment completes, broadcasting rounds can be run indefinitely to broadcast messages (even following server churn and elimination). In Figure 6-3, we report the latency of the broadcasting rounds on four different message sizes (ranging between 1 kB and 20 kB) and fraction of adversarial servers. The broadcasting rounds are between 10 and 120× faster compared to path establishment. With $f = 0.2$ and 100,000 users, broadcasting a 10 kB message incurs approximately four minutes of latency. We report micro-benchmarks for the

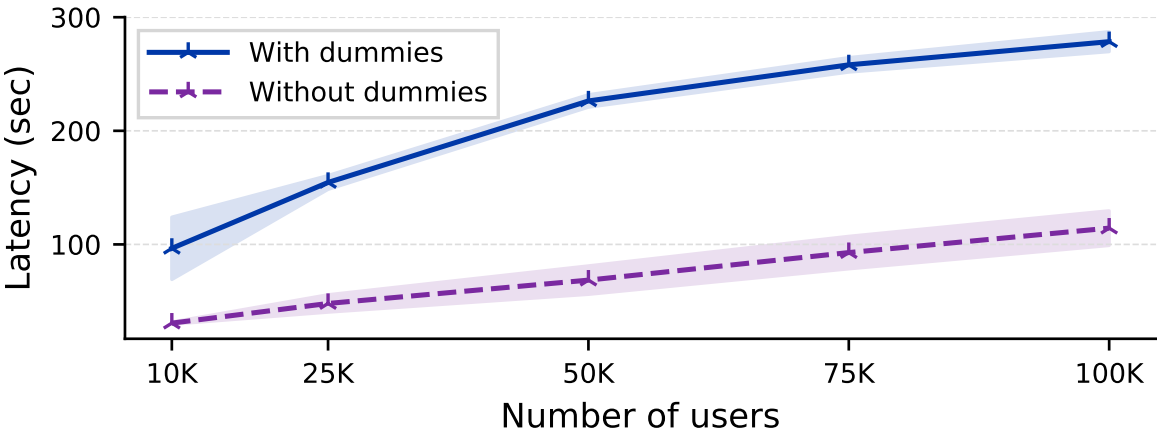

Figure 6-5: Dummy envelopes between layers incur a large overhead due to increased bandwidth requirements. Broadcast round (with and without dummy envelopes) evaluated on an international network (Oregon, Virginia, Frankfurt, and Stockholm) with 200Mbps bandwidth cap, $f = 0.2$, and message size set to 10 kB. Shaded regions represent a 95% confidence interval.

decryption and signing (performed by servers on each link):

- AES.Dec: 16 µs for 10 kB. // Envelope decryption.
- DS.Verify: 71 µs for 10 kB. // Signature verification.

Blame protocols. The primary computational overhead of blame protocols is generating a proof-of-decryption. In Trellis, this is done using a discrete-log equivalence proof (DLEQ), as explained in Chapter 4. We run a microbenchmark to determine the proving (and verification) time of each DLEQ:

- DLEQ.Prove: 152 µs. // Proving signing and correct decryption.
- DLEQ.Verify: 310 µs. // Proof verification done by server (or user).

The low overheads of the DLEQ proofs make our blame protocols lightweight. Similarly, arbitration (in case of a dispute) incurs one extra hop, which translates to an overall latency increase of 150 ms (in our network configuration).

6.3 Comparison to Atom

To the best of our knowledge, Atom [62] is the state-of-the-art system for metadata-private anonymous broadcast. Compared to Atom, Trellis achieves four orders of

magnitude lower latency (and proportionally higher throughput). Trellis scales easily to support large message sizes (e.g., in the kilobytes); in contrast, Atom does not, due to the zero-knowledge proofs and message passing within each anytrust group. We evaluate Atom on 32 B messages and multiply out to obtain the estimate time required to broadcast a 1 MB file (Kwon et al. [62] explicitly state that latency grows linearly with larger messages). We report the result for varying number of users and match Atom's $f = 0.2$ in Figure 6-6.

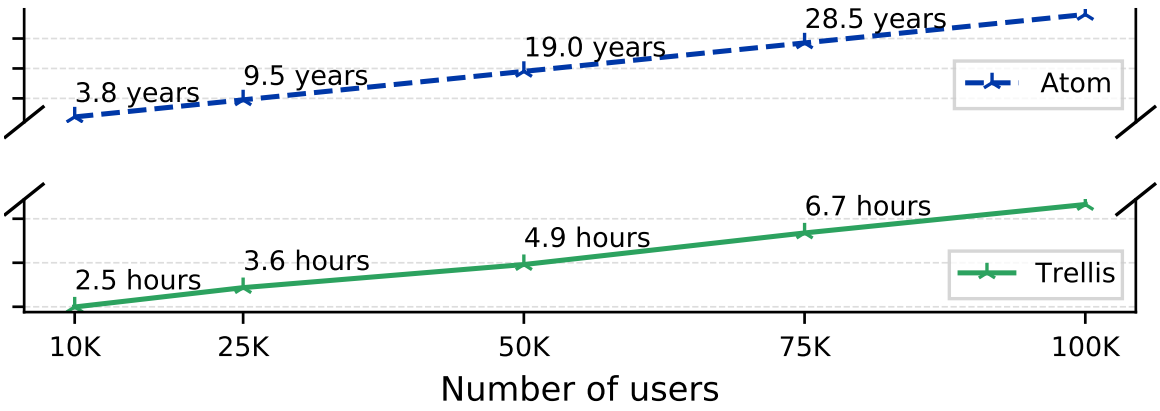

Figure 6-6: Latency of Trellis vs. Atom to broadcast a 1 MB file. We vary the number of users and set $f = 0.2$ for both systems (we run Atom on 32 B messages and extrapolate) with 200 Mbps bandwidth cap.

6.4 The price of metadata privacy

By decoupling path establishment and expensive operations from the message size, Trellis handles large messages better than all existing metadata-private anonymous broadcast systems, especially as the number of users increases. While Trellis is targeted to asynchronous applications (e.g., email, document uploads, surveys, etc.) due to concrete performance limitations (notably, high latency), further reducing latency and bandwidth overheads would require sacrificing security guarantees. By assuming that an adversary has a full view of the entire network, guaranteeing metadata privacy is imperative to preserving user anonymity. All metadata-private systems, including Trellis, will inherently require high bandwidth overheads so as to maintain a uniform view of all network traffic.

In Trellis, to achieve metadata privacy, we must introduce dummy envelopes between servers and fix the set of participating users. Our evaluation in Section 6.2 demonstrates that Trellis pushes metadata-private anonymous broadcast to its limit by being bandwidth—rather than computationally—bottlenecked. As a consequence, any further improvements in performance (beyond minor optimizations) will require sacrificing metadata privacy. Specifically, there are three possible security compromises that can be made: (1) removing dummy envelopes between layers, (2) allowing user churn, and (3) reducing the number of layers in the mixnet. All three of these compromises are made by systems like Tor so as to scale to millions of daily users and achieve *real-time* (synchronous) communication (e.g., web browsing). We briefly elaborate on the consequences of applying (1) and (2) on the practical security guarantees of Trellis. Applying (3) would result in messages not being mixed and therefore voids all guarantees.

- Not using dummy envelopes would maintain all the guarantees of Section 3.3, except for metadata privacy. In return, however, removing dummy envelopes would reduce the bandwidth overhead on the servers and improves the overall scalability of Trellis (see Figure 6-5). Of course, in practice it may be reasonable to assume a weaker adversary that is only capable of corrupting a subset of the servers and without a view of the entire network. Under this (weaker) threat model, the metadata privacy guarantee of Trellis can be sacrificed in favor of performance (without impacting the cryptographic anonymity guarantees of Trellis).
- Choosing to let users come and go makes Trellis (and all anonymous communication systems) vulnerable to intersection attacks [24, 25, 112]. Online (or offline) users correlate themselves with the message output patterns, making it possible to link messages back to the set of users that sent them. In practice, however, with sufficiently many users and a high churn rate (e.g., 50% of users churn in each round) the effectiveness of such attacks can be mitigated. Other methods, such as buddy sets [112], can also be applied to reduce the effectiveness of intersection attacks.

Chapter 7

Conclusion

Scalability and support for large messages plays a key role in many applications. For example, web browsing requires megabytes of data transfer (the median web page in 2021 was 2MB in size [99]). Additionally, in the real world, scalability and concrete efficiency play an important role in *anonymity* as well: the more users there are, the larger the anonymity set, and the more plausible deniability each user obtains.

Trellis is the first system to support large messages and horizontal scalability, advancing the state-of-the-art for anonymous broadcast. Our prototype of Trellis is potentially efficient enough to deploy at a small "enterprise" scale with a few hundred thousand users. In such a deployment, Trellis would incur a few hours of overhead to broadcast megabyte-sized files (e.g., PDF documents). There is still an efficiency gap to bridge in terms of performance when compared to systems like Tor and I2P, which provide weaker anonymity guarantees but better performance. However, Trellis is a step toward closing this gap.

Future work. The theoretical mixing analysis of Kwon [64] and our analysis in Section 3.9 assumes that the same number of envelopes are exchanged on each link, hence servers need to pad each envelope batch with dummy envelopes. One direction for improving the performance of Trellis would be to incorporate *different sized* batches into the mixing analysis and avoid adding dummy envelopes in Trellis (at the cost of increasing the number of layers). Doing so may reduce latency and improve the

horizontal scalability of Trellis, as illustrated in Figure 6-5. Another avenue could be proving a tighter bound on the number of layers required for mixing. Additionally, eliminating the need for digital signatures (currently required for our blame protocols) would concretely improve efficiency of mixing by a small factor, which may have a cumulative impact when the number of layers becomes very large or the network is very performant. For example, it is conceivable that message authentication codes (MACs) [11] could be suitable replacements to our digital signatures and result in smaller boomerang envelopes.

Bibliography

[1] Source code for Trellis. `https://github.com/SimonLangowski/trellis`, 2022.

[2] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder: Scalable, robust anonymous committed broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1233–1252, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417261. URL `https://doi.org/10.1145/3372297.3417261`.

[3] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

[4] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, 2017.

[5] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.

[6] Richard Arratia and Louis Gordon. Tutorial on large deviations for the binomial distribution. *Bulletin of mathematical biology*, 51(1):125–131, 1989.

[7] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Apostolos Pyrge-

lis, Bryan Ford, Joan Feigenbaum, and Jean-Pierre Hubaux. Prifi: Low-latency anonymity for organizational networks. *Proc. Priv. Enhancing Technol.*, 2020(4):24–47, 2020. doi: 10.2478/popets-2020-0061. URL `https://doi.org/10.2478/popets-2020-0061`.

[8] Ehrhard Behrends. *Introduction to Markov chains : with special emphasis on rapid mixing.* Advanced lectures in mathematics. Vieweg, Braunschweig/Wiesbaden, 2000. ISBN 3528069864.

[9] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. Var-CNN: A data-efficient website fingerprinting attack based on deep learning. *Proceedings on Privacy Enhancing Technologies*, 2019(4):292–310, 2019.

[10] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.

[11] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *https://crypto.stanford.edu/˜ dabo/cryptobook/BonehShoup_0_4. pdf*, 2017.

[12] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[13] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 92–102, 2007.

[14] Bryan Burrough, Sarah Ellison, and Suzanna Andrews. The Snowden saga: A shadowland of secrets and light. *Vanity Fair*, 2014. URL `https://www.vanityfair.com/news/politics/2014/05/edward-snowden-politics-interview`. Accessed March 2022.

[15] 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 *International conference on applied cryptography and network security*, pages 557–578. Springer, 2017.

[16] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[17] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. Talek: Private group messaging with hidden access patterns. In *Annual Computer Security Applications Conference*, pages 84–99, 2020.

[18] Cloudflare. Comprehensive DDoS protection. `https://www.cloudflare.com/ddos/`, 2021. Accessed May 2022.

[19] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 340–350. ACM, 2010.

[20] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[21] Cybersecurity and Infrastructure Security Agency. Security tip (ST04-015): Understanding denial-of-service attacks. `https://www.cisa.gov/uscert/ncas/tips/ST04-015/`, 2021. Accessed May 2022.

[22] Artur Czumaj and Berthold Vöcking. Thorp shuffling, butterflies, and non-Markovian couplings. In *International Colloquium on Automata, Languages, and Programming*, pages 344–355. Springer, 2014.

[23] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3-540-42580-2.

[24] George Danezis. Statistical disclosure attacks. In *IFIP International Information Security Conference*, pages 421–426. Springer, 2003.

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

[26] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *2003 Symposium on Security and Privacy, 2003.*, pages 2–15. IEEE, 2003.

[27] Debajyoti Das, Sebastian Meiser, Esfandiar Mohammadi, and Aniket Kate. Divide and funnel: a scaling technique for mix-networks. *Cryptology ePrint Archive*, 2021.

[28] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy Pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.

[29] DEDIS. Dedis advanced crypto library for Go. `https://github.com/dedis/kyber`, 2022. Accessed August 2022.

[30] Yvo Desmedt and Sushil Jajodia. Redistributing secret shares to new access structures and its applications, 1997.

[31] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[32] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[33] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1084–1101, 2019. doi: 10.1109/SP.2019.00050.

[34] Saba Eskandarian. Fast privacy-preserving punch cards. *Proceedings on Privacy Enhancing Technologies*, 2021(3):289–307, 2021. doi: doi:10.2478/popets-2021-0048. URL `https://doi.org/10.2478/popets-2021-0048`.

[35] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2022.

[36] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association. URL `https://www.usenix.org/conference/usenixsecurity21/presentation/eskandarian`.

[37] Nathan S Evans, Roger Dingledine, and Christian Grothoff. A practical congestion attack on Tor using long paths. In *USENIX Security Symposium*, pages 33–50, 2009.

[38] Ernesto Falcon. The FCC must update ISP privacy rules. `https://www.eff.org/deeplinks/2016/05/fcc`, 2016. Accessed May 2022.

[39] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 427–438. IEEE, 1987.

[40] Michael J Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 193–206, 2002.

[41] Nethanel Gelernter, Amir Herzberg, and Hemi Leibowitz. Two cents for strong anonymity: The anonymous post-office protocol. In *International Conference on Cryptology and Network Security*, pages 390–412. Springer, 2017.

[42] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *International*

Conference on the Theory and Applications of Cryptographic Techniques, pages 295–310. Springer, 1999.

[43] Yossi Gilad. Metadata-private communication for the 99%. *Communications of the ACM*, 62(9):86–93, 2019.

[44] Oded Goldreich. *Foundations of cryptography: a primer*, volume 1. Now Publishers Inc, 2005.

[45] Philippe Golle and Ari Juels. Parallel mixing. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 220–226, 2004.

[46] Maohua Guo and Jinlong Fei. Website fingerprinting attacks based on homology analysis. *Security and Communication Networks*, 2021, 2021.

[47] Stephen Hemminger et al. Network emulation with NetEm. In *Linux conf au*, volume 5, page 2005. Citeseer, 2005.

[48] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *annual international cryptology conference*, pages 339–352. Springer, 1995.

[49] Nicholas Hopper, Eugene Y Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)*, 13(2):1–28, 2010.

[50] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 233–253. Springer, 2014.

[51] Rebecca Jeschke. Internet advocates call on ISPs to commit to basic user privacy protections. `https://www.eff.org/deeplinks/2021/03/internet-advocates-call-isps-commit-basic-user-privacy-protections`, 2021. Accessed May 2022.

[52] Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (EdDSA). *RFC*, 8032:1–60, 2017.

[53] Antoine Joux. A one round protocol for tripartite diffie–hellman. In *International algorithmic number theory symposium*, pages 385–393. Springer, 2000.

[54] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[55] Aniket Kate and Ian Goldberg. Distributed private-key generators for identity-based cryptography. In *International Conference on Security and Cryptography for Networks*, pages 436–453. Springer, 2010.

[56] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2020.

[57] Andreas Klappenecker. Coupling of Markov chains. `https://people.engr.tamu.edu/andreas-klappenecker/csce658-s18/coupling.pdf`, 2018. Accessed May 2022.

[58] Ben Kreuter, Tancrède Lepoint, Michele Orrù, and Mariana Raykova. Anonymous tokens with private metadata bit. In *Annual International Cryptology Conference*, pages 308–336. Springer, 2020.

[59] Albert Kwon. Source code for Atom. `https://github.com/kwonalbert/atom`, 2017. Accessed August 2022.

[60] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 287–302, 2015.

[61] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proc. Priv. Enhancing Technol.*, 2016(2):115–134, 2016. doi: 10.1515/popets-2016-0008. URL `https://doi.org/10.1515/popets-2016-0008`.

[62] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 406–422, 2017.

[63] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable messaging system with cryptographic privacy. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 759–776, 2020.

[64] Young Hyun Kwon. *Towards anonymous and metadata private communication at Internet scale*. PhD thesis, Massachusetts Institute of Technology, 2019.

[65] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[66] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, 2018.

[67] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: Strong metadata security for voice calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 211–224, 2019.

[68] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. *ACM SIGCOMM Computer Communication Review*, 43(4):303–314, 2013.

[69] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity

networks. *ACM SIGCOMM Computer Communication Review*, 43(4):303–314, 2013.

[70] Stevens Le Blond, David 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 Conference on Special Interest Group on Data Communication*, pages 639–652, 2015.

[71] Brian N Levine, Michael K Reiter, Chenxi Wang, and Matthew Wright. Timing attacks in low-latency mix systems. In *International Conference on Financial Cryptography*, pages 251–265. Springer, 2004.

[72] Karsten Loesing, Steven J. Murdoch, and Roger Dingledine. A case study on measuring statistical data in the Tor anonymity network. In *Proceedings of the Workshop on Ethics in Computer Security Research (WECSR 2010)*, LNCS. Springer, January 2010.

[73] Donghang Lu and Aniket Kate. Rpm: Robust anonymity at scale. *Cryptology ePrint Archive*, 2022.

[74] Ewen MacAskill and Gabriel Dance. NSA files: decoded. *The Guardian*, 1, 2013.

[75] Nick Mathewson and Roger Dingledine. Practical traffic analysis: Extending and resisting statistical disclosure. In *International Workshop on Privacy Enhancing Technologies*, pages 17–34. Springer, 2004.

[76] Prateek Mittal and Nikita Borisov. Shadowwalker: peer-to-peer anonymous communication using redundant structured topologies. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 161–172, 2009.

[77] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy traffic analysis of low-latency anonymous communication

using throughput fingerprinting. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 215–226, 2011.

[78] Zachary Newman, Sacha Servan-Schreiber, and Srinivas Devadas. Spectrum: High-bandwidth anonymous broadcast. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 229–248, 2022.

[79] Lan Nguyen and Rei Safavi-Naini. Breaking and mending resilient mix-nets. In *International Workshop on Privacy Enhancing Technologies*, pages 66–80. Springer, 2003.

[80] Jonas Nick, Tim Ruffing, and Yannick Seurin. Musig2: simple two-round Schnorr multi-signatures. In *Annual International Cryptology Conference*, pages 189–221. Springer, 2021.

[81] NIST. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001. URL `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[82] Se Eun Oh, Nate Mathews, Mohammad Saidur Rahman, Matthew Wright, and Nicholas Hopper. Gandalf: Gan for data-limited fingerprinting. *Proc. Priv. Enhancing Technol.*, 2021(2):305–322, 2021.

[83] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[84] Lasse Overlier and Paul Syverson. Locating hidden servers. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–114. IEEE, 2006.

[85] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, 1991.

[86] Ania M Piotrowska. *Low-latency mix networks for anonymous communication*. PhD thesis, UCL (University College London), 2020.

[87] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The loopix anonymity system. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1199–1216, 2017.

[88] David Pointcheval and Jacques Stern. Provably secure blind signature schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 252–265. Springer, 1996.

[89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 73–85, 1989.

[90] Charles Rackoff and Daniel R Simon. Cryptographic defense against traffic analysis. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 672–681, 1993.

[91] Michael K Reiter and Aviel D Rubin. Anonymous web transactions with crowds. *Communications of the ACM*, 42(2):32–48, 1999.

[92] Eric Rescorla and Tim Dierks. The transport layer security (TLS) protocol version 1.3. *RFC*, 2018.

[93] Eric Rescorla and Tim Dierks. The transport layer security (TLS) protocol version 1.3. *RFC*, 2018.

[94] David Schatz, Michael Rossberg, and Guenter Schaefer. Hydra: Practical metadata security for contact discovery, messaging, and dialing. In *ICISSP*, pages 191–203, 2021.

[95] Adam Schwartz, Andrew Crocker, and Kit Walsh. EFF to court: Broadband privacy law passes first amendment muster. `https:`

//www.eff.org/deeplinks/2020/05/eff-court-broadband-privacy-law-passes-first-amendment-muster, 2020. Accessed May 2022.

[96] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In *European Symposium on Research in Computer Security*, pages 18–33. Springer, 2006.

[97] Robert Shostak, Marshall Pease, and Leslie Lamport. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3): 382–401, 1982.

[98] Payap Sirinam, Nate Mathews, Mohammad Saidur Rahman, and Matthew Wright. Triplet fingerprinting: More practical and portable website fingerprinting with n-shot learning. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1148, 2019.

[99] John Teague. 2021 Page Weight: The web almanac by HTTP Archive. https://almanac.httparchive.org/en/2021/page-weight, 2021. HTTP Archive.

[100] The Invisible Internet Project. I2P anonymous network, 2022. URL https://geti2p.net/en/. Accessed May 2022.

[101] The New York Times. Got a confidential news tip? https://www.nytimes.com/tips, 2022. Accessed May 2022.

[102] The Wall Street Journal. Got a tip? https://www.wsj.com/tips, 2022. Accessed May 2022.

[103] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.

[104] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A Wood. A fast and simple partially oblivious PRF, with applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 674–705. Springer, 2022.

[105] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152, 2015.

[106] Von Spiegel Staff. Inside the NSA's war on internet security. *Der Spiegel*, 2014. URL `https://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html`. Accessed March 2022.

[107] Abraham Waksman. A permutation network. *Journal of the ACM (JACM)*, 15(1):159–163, 1968.

[108] Jun Wan, Hanshen Xiao, Elaine Shi, and Srinivas Devadas. Expected constant round byzantine broadcast under dishonest majority. In *Theory of Cryptography Conference*, pages 381–411. Springer, 2020.

[109] Douglas Wikström. Five practical attacks for "optimistic mixing for exit-polls". In *International Workshop on Selected Areas in Cryptography*, pages 160–174. Springer, 2003.

[110] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.

[111] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.

[112] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. Hang with your buddies to resist intersection attacks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1153–1166, 2013.

[113] Theodore M. Wong, Chenxi Wang, and Jeannette M. Wing. Verifiable secret

redistribution for threshold sharing schemes. Technical report, Carnegie Mellon University, 2002.

[114] Shoko Yonezawa, Tetsutaro Kobayashi, and Tsunekazu Saito. Pairing-friendly curves. *Network Working Group. Internet-Draft. January*, 2019.