ZXAD: Zero-knowledge Exit Abuse Detection for Tor

Akshaya Mani
University of Waterloo
akshaya.mani@uwaterloo.ca

Ian Goldberg
University of Waterloo
iang@uwaterloo.ca

Abstract

The Tor anonymity network is often abused by some attackers to (anonymously) convey attack traffic. These attacks abuse Tor exit relays (*i.e.*, the relays through which traffic exits Tor) by making it appear the attack originates there; as a result, many website operators indiscriminately block all Tor traffic (by blacklisting all exit IPs), reducing the usefulness of Tor.

Recent research shows that majority of these attacks are ones that generate high traffic volume (*e.g.*, Denial-of-Service attacks). This suggests that a simple solution such as throttling traffic flow at the Tor exits may permit early detection of these attacks.

However, naïvely monitoring and throttling traffic at the Tor exits can endanger the privacy of the network's users. Indeed, many recent works have proposed private measurement systems that support *safe* aggregation of exit statistics. However, these systems do not permit identification of "unlinkable" connections that are part of a high-volume attack. Doing so could allow Tor to take proper remedial actions, such as dropping the attack traffic, but care must be taken to protect privacy.

We present ZXAD (pronounced "zed-zad"), the first zero-knowledge based *private* Tor exit abuse detection system. ZXAD detects large-volume traffic attacks without revealing any information, apart from the fact that some user is conveying a high volume of traffic through Tor. We formally prove the correctness and security of ZXAD. We also measure two proof-of-concept implementations of our zero-knowledge proofs and show that ZXAD operates with low bandwidth and processing overheads.

1 Introduction

Tor [23] is used by millions of people daily for anonymous communication over the Internet. While Tor has many legitimate uses such as whistleblowing, censorship avoidance, and protecting one's security and privacy online [51], it is often abused by some attackers to (anonymously) convey attack traffic supporting spam campaigns, vulnerability scanning, content scraping, *etc.* [44].

Since all traffic exits Tor through a set of publicly listed special relays, called the Tor *exits*, these relays often end up being blacklisted for all the malicious (or objectionable) content routed through Tor. This acts as strong disincentive, discourages volunteers from running exit nodes, and hinders the growth of a volunteer-operated network like Tor.

Moreover, even the actions of a single malicious user can trigger the automated abuse detectors at the destination web servers, cause content providers to blacklist the exit, and in turn block all users (including legitimate users) connecting through that exit; *i.e.*, *fate-sharing*. Some website operators even go to the extent of providing differential treatment to all Tor users — ranging from solving simple CAPTCHAs to outright blocking [34]. As a result, over time the entire network may become unusable. Abuse of Tor exits therefore tends to be one of the greatest threats against the growth of Tor.

Recent research by Singh et al. [48] shows that a majority of the attacks originating from Tor are those that generate high traffic volume (*e.g.*, SSH brute-force or Denial-of-Service attacks). This suggests that a simple solution, such as rate limiting the number of connections allowed per client to a target destination at the exits, could potentially stop most of these attacks. For instance, the threshold number of connections can be set to 1 for an SSH connection, 10 for websites whose resources appear on multiple pages, or unlimited for very popular destinations such as facebook.com or google.com.

However Tor's protections make rate limiting challenging: (i) multiple connections from the same client are supposed to be *unlinkable*, yet in order to limit per-client connections, the exits must be able to identify connections coming from the same client and (ii) naïvely monitoring and throttling attack traffic using an Intrusion Detection System (IDS), such as Zeek [60] or Suricata [43], at the Tor exits can pose significant *privacy* risk to the network's users. Besides, monitoring users' communications is antithetical to the Tor Project's goals.

To address all these challenges, we introduce ZXAD (pro-

nounced "zed-zad"), a zero-knowledge based exit abuse detection system for Tor. ZXAD incurs low bandwidth and moderate processing overheads, and does not require any significant changes to Tor's existing design. The goal of ZXAD is to detect large-volume traffic (from an individual Tor client) destined to a target server, in a privacy-preserving way. That is, ZXAD does not reveal any information other than the fact that some Tor client is making numerous connections to a target destination. ZXAD provides just enough information for the exits to take proper remedial actions (*e.g.*, dropping attack traffic).

ZXAD achieves these goals by providing a virtual token dispenser to each Tor client that allows the client to dispense at most n anonymous and *unlinkable* tokens per ZXAD epoch for every destination. The value of n is set based on the popularity of the destination (*e.g.*, 1 for port 22). The client uses these tokens to authenticate to the Tor exits every time it makes a connection to a new destination using the same exit or the same destination using a different exit. This way a client has to "double-spend" (or re-use) a token to make more than n connections to a target destination, linking them.

Malicious clients can still try to connect using different exits — but then the Tor exits can forward the token to the target destination, which has a "global view" to rate limit double-spending clients. Importantly, these tokens are unlinkable; *i.e.*, given two different tokens neither the Tor exit nor the destination server can tell if the tokens were from the same Tor client or not. Therefore, the server and the exit learn no other information apart from the fact that some client is double-spending.

In the following sections, we introduce ZXAD, formally prove its correctness and security, implement our zero-knowledge proofs, and demonstrate that ZXAD operates with low bandwidth and processing overheads.

2 Background

In this section, we present a brief overview of Tor and review some well-known cryptographic primitives and protocols that are used as building blocks in the construction of ZXAD.

2.1 Tor

Tor [23] provides anonymity by relaying traffic via "anonymous paths", called *circuits*, that are constructed by randomly selecting multiple (usually three) relays. Along the path, layered encryption is used to conceal the actual sender (*i.e.*, the Tor client) and the receiver (*i.e.*, the destination), so that each relay knows only the previous hop and the next hop. Traffic flows down the circuit in fixed-size *cells* carrying encrypted routing information and data [22].

The first relay in a circuit is usually the *guard*, a Tor relay that is relatively stable, fast, and reliable. The next hop, the *middle* relay, relays traffic from the guard to the final relay, called the *exit*, which finally establishes a TCP connection to the intended destination. Since traffic exits the Tor network through the exit relays, they are often blamed when something malicious or objectionable is routed through them.

Tor maintains long-standing TLS connections between relays that are adjacent on some Tor circuit. Communications over different circuits that share a hop between two relays are sent over the same TLS connection. A Tor *stream* is analogous to a regular TCP connection between the Tor client and a target destination. Several *streams* may be multiplexed over the same circuit. The Tor client usually switches to a new circuit every ten minutes.

To ensure all Tor clients have the same "view" of the Tor network, the Tor directory authorities (*DirAuths*), a set of nine dedicated servers, periodically publish a *consensus* document, containing information on all currently running relays that make up the Tor network. The consensus is reached using the Tor directory protocol [50], a majority voting protocol which makes sure that only "updates" signed by a majority (at least five) of the authorities are added to the consensus document.

Creating Tor circuits. A Tor client maintains a single connection to each of its guards, through which multiple circuits may be created. To begin creating a new circuit, the client first sends a *create* cell to the Tor guard in the chosen path, initiating a Diffie-Hellman handshake. The guard then responds with a *created* cell, completing the handshake and the first hop of the circuit. Next, to extend the circuit one hop further, the client sends a relay *extend* cell to the guard, specifying the address of the middle node, and the Diffie-Hellman handshake for the middle node. On receiving the relay extend cell, the guard copies the handshake into a create cell, and passes it to the middle node to extend the circuit. The middle node then responds with a created cell to the guard, which then encrypts the payload into a relay *extended* cell and passes it back to the client. Finally, to extend the circuit to a third hop (usually an exit relay), the client informs the middle relay to extend the circuit one hop further, which proceeds in a similar way as above, and the circuit is complete. Once the Tor client has established the circuit, it sends *begin* cells to create streams to a specified destination server and port, and *relay data* cells that carry end-to-end stream data.

Tor Browser. The Tor Browser first creates a new circuit for each unique domain entered by the user in the browser address bar. It then creates a new stream over this circuit for retrieving the web page. Subsequent streams that are created for fetching the embedded resources, such as images, scripts, *etc.*, are multiplexed over the same circuit.

2.2 Preliminaries

We now briefly review some concepts and background that are necessary for understanding ZXAD.

Decisional Diffie-Hellman assumption. Let $\mathbb{G}$ be a cyclic

multiplicative group of prime order q and g be one of its generators. The Decisional Diffie-Hellman (DDH) assumption [6] states that it is computationally hard to distinguish between the two distributions $\langle g^a, g^b, g^{ab} \rangle$ and $\langle g^a, g^b, g^c \rangle$ where a, b, c are drawn uniformly at random from $\mathbb{Z}_q^*$.

Bilinear groups. Let $\mathcal{G}$ be an asymmetric bilinear group generator that takes as input a security parameter 1^k and returns a tuple $\Lambda = \langle q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, H_1, H_2 \rangle$ where $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_t$ are cyclic multiplicative groups of prime order q, $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_t$ is an efficient and non-degenerate bilinear map, g_1 and g_2 are generators of the groups $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively, and $H_1 : \{0,1\}^* \rightarrow \mathbb{G}_1$ and $H_2 : \{0,1\}^* \rightarrow \mathbb{G}_2$ are hash functions that map binary strings to elements of $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively.

ZXAD uses Type III pairings [25]: $\mathbb{G}_1 \neq \mathbb{G}_2$ and there exists no efficiently computable homomorphisms between $\mathbb{G}_1$ and $\mathbb{G}_2$. In other words, the Symmetric eXternal Diffie-Hellman (SXDH) assumption holds; *i.e.*, the DDH assumption holds in both $\mathbb{G}_1$ and $\mathbb{G}_2$.

BLS signature. A primary primitive used in our construction is the BLS signature [7]. We use the notation BLS to define a variant with public key in $\mathbb{G}_1$ and signature in $\mathbb{G}_2$.

Let $\Lambda = \langle q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, H_1, H_2 \rangle$ be the output of an asymmetric bilinear group generator. In BLS, a keypair (v, V) is generated by choosing a private key $v \xleftarrow{R} \mathbb{Z}_q$ and setting the public key to $V = g_1^v$. A message $M \in \{0,1\}^*$ is signed by producing the signature $\sigma = H_2(M)^v$. A signature σ on message M is valid if and only if $e(g_1, \sigma) \stackrel{?}{=} e(V, H_2(M))$.

Zero-knowledge proofs. Zero-knowledge proofs (ZKPs) [26] limit the amount of information transferred between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ in a cryptographic protocol. Throughout this paper, we make use of the Generalized Schnorr Proofs (GSPs) introduced by Camenisch and Stadler [12] and formally defined by Camenisch et al. [14] to prove knowledge and relationships of discrete logarithms. The Σ-*protocol* for such proofs are usually defined as a three-phase interactive protocol. Non-interactive versions of such proofs can be obtained using the Fiat-Shamir heuristic [24].

A zero-knowledge proof satisfies the following three properties: (i) *Completeness* guarantees that a valid proof will always be accepted by the verifier; (ii) *Soundness* guarantees that only a valid proof will be accepted by the verifier; and (iii) *Zero-knowledgeness* guarantees that a valid proof does not reveal anything about the witnesses.

zkSNARKs. Informally, a Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARK) [5] is a proof construction, where one can prove the truth of a statement without revealing any information (besides the fact that the statement is true), and without any interaction between the prover and verifier. Additionally, it satisfies the *succinctness* condition; *i.e.*, the proof size and verification time are constant even for arbitrarily large statements.

Shamir's secret-sharing scheme. A *threshold* based secret-sharing scheme by Shamir [47] allows subsets of t or more parties to recover a split secret. It distributes the secret using a $t-1$ degree polynomial and is based on the idea that at least t points are required to reconstruct a polynomial of degree $t-1$. Therefore, no group of fewer than t parties can reconstruct the secret.

3 Overview

We first present the current state of Tor exit abuse detection and mitigation. Next, we introduce ZXAD by describing an IP-based credential issuing service (for use mainly by ZXAD), the participants, threat model, the system model, and the different phases of the protocol.

3.1 Abuse of Tor exits

Tor currently does not have any built-in mechanisms to detect malicious users that abuse benign Tor exits by conveying attack traffic. A malicious Tor user can abuse Tor exits in three ways: (i) *circuit level* — by sending malicious stream(s) within a single circuit, (ii) *exit level* — by generating multiple malicious circuits through a single exit, and (iii) *Tor level* — by generating malicious circuits through multiple exits.

The circuit-level attacks are the easiest to mitigate — the exits can just rate limit the number of streams per circuit. However, the exit-level and Tor-level attacks are much harder to mitigate since exits cannot perceive different circuit connections that are a part of large-volume attack (by a single Tor client) as connections coming from the same client. This is because the traffic from different circuit connections are *unlinkable* — given two Tor circuits, the exits cannot tell if the circuits originate from the same Tor client or not.

Currently, opportunistic onions [46] by Cloudflare is the closest solution to evade exit-level and Tor-level attacks. It relies on malicious clients repeating the time-consuming "rendezvous protocol" to rate limit the number of circuits created by each Tor client. However, their solution only works for websites hosted by Cloudflare (*i.e.*, $\sim$ 10% of the Internet [45]), it does not stop exits from being abused or service providers (at large) from blocking Tor. Section 4.2 describes how ZXAD can be used to mitigate exit-level and Tor-level attacks.

3.2 IP-based credential service

Fundamentally, ZXAD requires *some* way to distinguish many users making one connection each to some particular webserver from one user making many connections (a Sybil attack).

Many previous related research works [28; 29; 31; 54; 55], as well as common deployed network services, limit clients

based on their IP address. However, using IP address as a client identifier has the major limitation that it is neither permanent nor unique [27] (*e.g.*, mobile clients with dynamic IP addresses, clients behind a Network Address Translation (NAT), *etc.*). Nonetheless, IP address is a ubiquitous identifier websites use today outside of Tor to block abusers; therefore we allow the use of an IP address as the *client identifier* in ZXAD as well. In any case, ZXAD is sufficiently general to be adapted for any type of 'one-per-person' identifier, such as a government-issued ID [27] (*e.g.*, e-Passport, enhanced driver's license, *etc.*), a valid X.509 certificate chain, and so on. ZXAD does not require the identifier to be high entropy, and where possible, will allow the client to prove its possession in zero knowledge [21].

In order to prove possession of an IP address, however, the Tor client must make a direct connection using that IP address. This can potentially deanonymize the Tor client if this connection can be linked to the sites it visits over Tor. Therefore, we introduce an IP-based credential issuing service from which the client can obtain an IP-based credential and prove possession of the same (in zero-knowledge) to ZXAD entities. The client uses this credential as its verifiable identifier (or long-term key) in the ZXAD protocol. However, in the case of client identifiers that can change over time, such as the IP address, this credential must be refreshed based on the expected lifetime of the identifier (*e.g.*, monthly). In other words, every month the credential issuer regenerates fresh signing key(s) and all clients obtain a new credential. In this case, the Sybil attacks are limited to how many identifiers the attacker can control in a month (*i.e.*, a single long-term key lifetime).

The IP-based credential issuing service can be run by any set of honest-majority servers (*e.g.*, the DirAuths in Tor). Note that if the DirAuths were to act as the credential issuers, a malicious DirAuth can learn the list of all users requesting a credential. However, we envision that the IP-based credential issuing service could be useful for (anonymously) proving possession of an IP to a variety of services through any anonymizing network. In such a case, many more people could be using this service (not just the users of Tor or ZXAD) and hence it is justified to be an independent or standalone service by itself. For now, however, we ignore this wide applicability and prove the security of ZXAD, even in a setting where the DirAuths are the credential issuers.

3.3 Participants and threat model

ZXAD is a distributed system that relies on multiple entities to achieve its privacy and security goals. The participants of the system are the Tor clients, the nine DirAuths, the exits, and the end server. We now detail the trust assumptions on the different entities of ZXAD.

Tor clients. The Tor clients can behave maliciously: (i) they can try to make more than the allowed number of connections to a target website or (ii) they can try to impersonate other honest clients such that an honest client's connections are rate-limited by the target destination before the threshold is reached. As mentioned in Section 3.2, ZXAD uses IP address as the *client identifier* to detect such abuse by a Tor client.

Malicious clients can also cause a denial-of-service by submitting far too many long-term and periodic key (described in Section 3.4 below) requests to the DirAuths. However, the DirAuths can rate-limit these requests without affecting honest clients much (for details see Section 9).

DirAuths. As with Tor, in ZXAD we assume that at least five out of the nine DirAuths are honest. If a majority of the DirAuths are compromised, then the DirAuths can even compromise Tor, let alone relay attack traffic through exit nodes by exploiting ZXAD.

In ZXAD, any malicious DirAuth learns the IP addresses of all ZXAD users when they obtain or refresh long-term keys. However, if the IP-based credential service is run as a standalone separately useful entity, we envision that more people would be using this service (not just the users of Tor or ZXAD), which will mitigate this effect. In any case, malicious DirAuths do not have enough information to link any connection to a specific Tor user even when they collude with malicious exits (or end servers).

Exit nodes and servers. Malicious exits (and end servers) can collude with malicious DirAuths and try to deanonymize a user. However, ZXAD guarantees that no information about an honest client is ever leaked.

In short, ZXAD is secure as long as a majority of the DirAuths are honest. Informally, the privacy guarantees of ZXAD are (i) A set of honest exits (collectively) will accept no more than n_s connections (to a target destination $\mathcal{S}$) from a single Tor client in any given epoch and (ii) no information about the identity of an honest Tor client is ever exposed. We provide the formal security definitions and proof in Section 5.

3.4 System model

The workflow of ZXAD begins when a Tor client joins the Tor network for the first time. During the bootstrapping process, the client directly connects to the DirAuths and obtains a *deterministic* signature on its IP address. The client uses this signature as its long-term key in ZXAD. Note that the long-term key is never revealed to any ZXAD entity (not even the DirAuths).

After joining the network, the client sends its blinded long-term key to the DirAuths. The DirAuths first verify if the blinded key is valid and then perform a 'blind signature transfer'; *i.e.*, issue another deterministic signature on the same IP address embedded in the long-term key. Unlike the long-term key, this signature is valid only for a short period (*e.g.*, a week) and hence called the periodic key. The client updates its periodic key every period (*e.g.*, weekly).

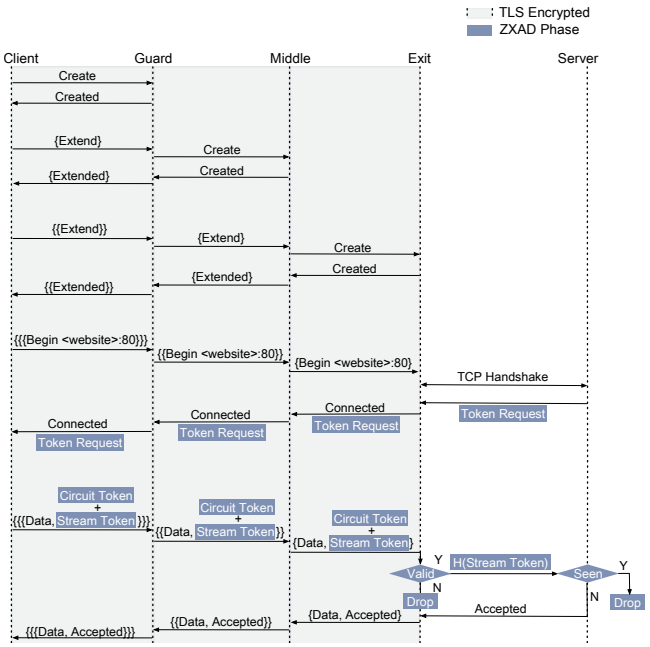

Figure 1: The flow diagram of the Tor client connecting to a website under a large-volume attack (via a three-hop circuit) — the shaded boxes represent different ZXAD phases.

ZXAD operates in 10-minute epochs. Figure 1 illustrates how the different phases of ZXAD are performed when a Tor client connects to a connection-throttling destination via a three-hop circuit. As shown in Figure 1, the destination requests tokens only when it is facing a large-volume attack from Tor (see Section 9 for implementation details). When the destination requests a token, the client (generates and) sends an unlinkable and unique stream token and a zero-knowledge proof (that the stream token is well-formed) to the exit node. Recall that the maximum number of *unlinkable* connections allowed per epoch, n_S, to a target destination $\mathcal{S}$ is preset based on the popularity of the destination (*e.g.*, 1 for port 22). Therefore, the Tor client can create at most n_S unique stream tokens per epoch for every destination $\mathcal{S}$ and must (linkably) open multiple streams in a single circuit to make more than n_S connections. On receiving the stream token and the zero-knowledge proof, the exit accepts the token if and only if (i) the proof is valid and (ii) it has not previously seen this token in the current epoch. The latter check ensures that the client does not "double spend" a token.

In simple terms, ZXAD provides a virtual token dispenser (to each Tor client) that allows the client to dispense at most n_S anonymous and unlinkable tokens per epoch for every destination $\mathcal{S}$.

3.5 ZXAD Phases

The ZXAD protocol has seven phases (the post-initialization phases are illustrated in Figure 2):

Initialization phase. The initialization phase involves the configuration of system parameters and the generation of threshold 5-out-of-9 long-term and periodic signing keys for the DirAuths.

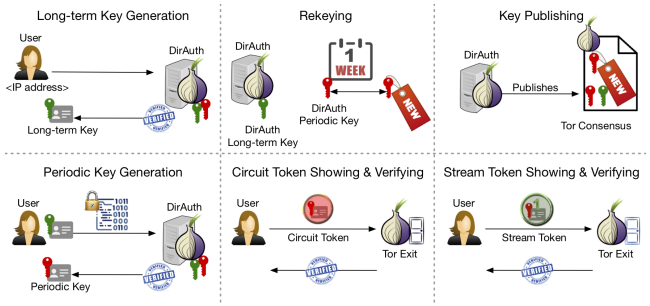

Figure 2: An overview of the post-initialization phases in the ZXAD protocol in a single-DirAuth setting.

The long-term signing key is comparable to the long-term identity keys in Tor [23], which are usually never changed. However, *only* when using client identifiers that can change over time (*e.g.*, the IP address), the long-term signing key must be regenerated based on the expiry of the client identifiers (as mentioned in Section 3.2) by running a rekeying phase similar to the *periodic rekeying phase* described below. For simplicity, we present the long-term signing keys as static in the rest of the paper.

We use additive secret sharing [4] to generate additively shared long-term and periodic signing keys once at the initialization phase, and then use share conversion [18] to turn them into a Shamir-shared key. This way, the later rekeying phases can be performed locally, without any communication among the DirAuths. We describe this key generation protocol in Section 4.3.

Long-term key generation phase. In the the long-term key generation phase, the client obtains its deterministic long-term key from the DirAuths. In the case where the IP-based credential issuing service is a standalone entity, the client just uses its IP-based credential as the long-term key (as mentioned in Section 3.2).

This step is done so that during every run of the periodic rekeying phase, the DirAuths just verify a signature (with their joint long-term secret key) on the client's IP address, rather than verifying the client IP address itself.

Rekeying phase. In the (unlikely) event where a majority of the DirAuths' periodic secret keys are compromised in the same period, the adversary would be able to learn the client's identity by performing a brute-force attack over all possible IP addresses.

Therefore to limit the amount of time a periodic key is vulnerable, the DirAuths periodically compute a new shared secret signing key (and the associated joint public key) for the next period, in a forward-secret manner.

Note that the period duration can be anywhere between a day and several months. A small duration (such as a day) would increase the load on the DirAuths as they would need

to issue a periodic key to each Tor user every day. Similarly, a longer duration (such as several months) would increase the amount of exposure in the unlikely case where a majority of the DirAuths' periodic secret keys are compromised. Therefore, to moderate the performance and security risks involved, we suggest a reasonable default period of one week. Throughout the paper, we refer to the period as the ZXAD *period* (or the period in general).

As described below in the *key publishing phase*, the clients and the exits use the Tor's existing consensus protocol to obtain copies of the DirAuths' public key. Additionally, every Tor client obtains a periodic key (*i.e.*, runs the *periodic key generation phase*) once per period, since the DirAuths' joint key has changed.

Key publishing phase. The key publishing phase is performed once, at the beginning of every ZXAD period. The DirAuths publish the current and the next period public keys in the Tor consensus.

The Tor clients and the exits obtain the DirAuths' public key when they update their consensus. Note that the exits always obtain the current period's public key while the clients obtain the current (or the next period's) public key depending on which keypair is used to generate the periodic key (in the *periodic key generation phase* described below).

Periodic key generation phase. In the periodic key generation phase, the client obtains the periodic key (for the current or the next ZXAD period), by specifying which keypair the DirAuths must use for generating the periodic key. Although the client can obtain both the current and the next period's key (in the current ZXAD period itself), it can only use the current periodic key to generate the current period's circuit and stream tokens (described below). However, we still recommend that a client obtains its periodic key in advance to evade deanonymization by malicious DirAuths using traffic correlation [33]. We note that as long as the client does not use Tor right after obtaining the current period's key from the DirAuths, it is not prone to such deanonymization attacks.

In short, this phase produces a deterministic unique short-term identifier that can be used to produce the stream and circuit tokens (described below) that are globally unique to a given long-term key (or an individual client when using 'one-per-person' identifiers). The tokens produced are unlinkable; *i.e.*, given two different tokens the Tor exit (or the destination server) cannot tell if the tokens were from the same Tor client or not.

Circuit token showing and verification phase. A client sends a circuit token and a zero-knowledge proof along with the *first* stream token (within a circuit) to the exit. These tokens and proofs can be computed offline in advance. This proof ensures that the client possesses a credential on the IP address (embedded in the token) used to create the periodic key, without revealing it. The circuit token and proof is purely for optimization purposes and reduces the verifying time at the exit for subsequent stream tokens sent within the circuit.

Stream token showing and verification phase. As mentioned before, ZXAD operates in 10-minute *epochs*, in order to limit the amount of time a stream token is usable by a Tor client. This is comparable to the default circuit lifetime of ten minutes in Tor. Throughout this paper, we refer to this ten-minute period as a ZXAD *epoch* and this should not be confused with the hour-long Tor epoch used to create consensuses.

Every time the client creates a stream to a new destination under a large-volume attack within any given circuit, it sends a stream token and a zero-knowledge proof to the exit that the token is valid. The Tor exit accepts the token if and only if the proof is valid and it has not seen that stream token before in the ZXAD epoch (in which the associated circuit was created). Otherwise, the exit terminates the circuit.

4 Protocol Details

We first describe a basic version of the ZXAD protocol in a single-DirAuth setting, which is easy to understand. Next, we describe how ZXAD can be used to combat exit-level and Tor-level attacks. Finally, we extend the basic ZXAD protocol to a t-out-of-n DirAuths setting (in Section 4.3) and describe the changes pertaining to handling distributed DirAuths in each of the phases.

4.1 Basic ZXAD Protocol

We now describe a basic version of the ZXAD protocol in a setting with a single DirAuth, which is clear and easy to understand.

Initialization phase. Let $\Lambda = \langle q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, H_1, H_2 \rangle$ be the output of an asymmetric bilinear group generator and let $Y_1 \xleftarrow{R} \mathbb{G}_1$ be a public parameter. Let $\langle \rho, P = g_1^{\rho} \rangle$ and $\langle \alpha, A = g_1^{\alpha} \rangle$ be the long-term and the periodic keypair of DirAuth $\mathcal{A}$.

Long-term key generation phase. Consider a Tor client C. Let IP_C be the IP address of client C. During the bootstrapping process, the Tor client directly connects to the DirAuth $\mathcal{A}$ and obtains its long-term key $\sigma_{\mathcal{P}} = H_2(IP_C)^{\rho}$ (a `BLS` signature) from $\mathcal{A}$.

Key publishing and Rekeying. At the beginning of every period, the DirAuth creates the secret signing key α and public key g_1^{α} for the *next* period, and publishes the current and the next period public keys in the Tor consensus. The Tor clients and the exits obtain the DirAuths' public key when they update their consensus.

Periodic key generation phase. In the the periodic key generation phase, C first specifies which keypair the DirAuth should use (*i.e.*, the current or the next ZXAD period's) and obtains a blind BLS signature [7] on its IP address from

the DirAuth $\mathcal{A}$. The client uses this signature as its periodic key (for the corresponding period) in the remainder of the protocol.

To receive its blind signature, C first computes $B = H_2(IP_C)$. C then blinds B and its long-term key $\sigma_{\mathcal{P}}$ by choosing random $r_C \xleftarrow{R} \mathbb{Z}_q^*$ and setting $\widetilde{B} = B^{r_C}$ and $\widetilde{\sigma}_{\mathcal{P}} = \sigma_{\mathcal{P}}^{r_C}$. Finally, C sends $\langle \widetilde{B}, \widetilde{\sigma}_{\mathcal{P}} \rangle$ through one of its guards to the DirAuth $\mathcal{A}$. Note that blinding B prevents the DirAuth and the guard from learning B and $\sigma_{\mathcal{A}}$ (below).

On receiving the blinded hash $\widetilde{B}$ of the client's IP address and the blinded BLS signature $\widetilde{\sigma}_{\mathcal{P}}$, the DirAuth $\mathcal{A}$ verifies if $e(g_1, \widetilde{\sigma}_{\mathcal{P}}) \stackrel{?}{=} e(P, \widetilde{B})$ and performs a BLS blind signature transfer by computing $\widetilde{\sigma}_{\mathcal{A}} = \widetilde{B}^{\alpha}$. That is, if the client already possesses a valid signature under ρ on some value IP_C, then $\mathcal{A}$ issues the client a BLS blind signature under α on the same value. Note that, the DirAuth does not learn IP_C or its hash $B = H_2(IP_C)$. $\mathcal{A}$ then sends $\widetilde{\sigma}_{\mathcal{A}}$ back to C through the same guard.

Note that the client need not separately prove knowledge of r_C, since $\widetilde{\sigma}_{\mathcal{A}}$ is useless without the correct r_C value. Therefore, the client can choose some random $r_C \xleftarrow{R} \mathbb{Z}_q^*$, compute $\widetilde{B}$ and $\widetilde{\sigma}_{\mathcal{P}}$ offline (any time after getting the long-term key), and even discard the r_C values.

On receiving $\widetilde{\sigma}_{\mathcal{A}}$, C verifies if $e(g_1, \widetilde{\sigma}_{\mathcal{A}}) \stackrel{?}{=} e(A, \widetilde{B})$ and aborts if not. This check ensures that the DirAuth's BLS signature on $\widetilde{B}$ is valid. C then unblinds $\widetilde{\sigma}_{\mathcal{A}}$ and obtains $\sigma_{\mathcal{A}} = (\widetilde{\sigma}_{\mathcal{A}})^{1/r_C} = B^{\alpha}$, the DirAuth's BLS signature on B (which is the same B embedded in the long-term key). Note that the client can of course unblind wrongly to produce a valid signature on some other random (unknown) hash value, but then the BLS signature here will not match the circuit token proof (defined below), and the client's circuit tokens will not verify.

Observe that $\sigma_{\mathcal{A}}$ is a deterministic function of IP_C and α and has high entropy and so cannot be brute-forced. Therefore, we use $\sigma_{\mathcal{A}}$ to produce randomized circuit and deterministic stream tokens (defined below) that are bound to the client C.

The client obtains a new periodic key every ZXAD period (at most one period in advance) when the DirAuth's signing key changes (*i.e.*, after every rekeying).

Circuit token showing and verification phase. At this point C has obtained a blind signature $\sigma_{\mathcal{A}}$ from $\mathcal{A}$, has an active circuit through some exit, and has established a connection to some connection-throttling destination server-port combination, represented as $\mathcal{S}$. If $\mathcal{S}$ is the *first* destination (within the circuit) requesting for a ZXAD token, then C produces a circuit token T_c and a zero-knowledge proof Π_{T_c} as follows:

1. *Token:* The token T_c is a randomized commitment to the periodic key $\sigma_{\mathcal{A}}$. To generate the token, C chooses $r_2 \xleftarrow{R} \mathbb{Z}_q^*$, computes $g_2'' = g_2^{r_2}$ and $\sigma_{\mathcal{A}}'' = \sigma_{\mathcal{A}}{}^{r_2}$, and sets $T_c = \langle g_2'', \sigma_{\mathcal{A}}'' \rangle$. Note that the client can compute the circuit tokens offline (any time after getting the new periodic key) by just choosing some random $r_2 \xleftarrow{R} \mathbb{Z}_q^*$.
2. *Zero-Knowledge Proof:* C constructs a non-interactive zero-knowledge proof Π_{T_c} (that T_c is *well formed*):
$$\Pi_{T_c} = PK\Big\{(r_2, \sigma_{\mathcal{A}}, B, IP_C) : [g_2'' = g_2^{r_2}] \bigwedge [\sigma_{\mathcal{A}}'' = \sigma_{\mathcal{A}}{}^{r_2}] \bigwedge [sig_{\mathcal{A},A}(B) = \sigma_{\mathcal{A}}] \bigwedge [B = H_2(IP_C)]\Big\}$$
where $sig_{\mathcal{A},A}(B) = \sigma_{\mathcal{A}}$ means that $\sigma_{\mathcal{A}}$ is a valid BLS signature on message B (with B already hashed, as $B = H_2(IP_C)$) by the DirAuth $\mathcal{A}$ with the secret key corresponding to the public key A.

The proof has three parts: First it proves that the client knows some r_2 such that $g_2'' = g_2^{r_2}$ and $\sigma_{\mathcal{A}}'' = \sigma_{\mathcal{A}}{}^{r_2}$, for some $\sigma_{\mathcal{A}}$. Next, it proves $\sigma_{\mathcal{A}}$ is a valid signature on some B by the DirAuth $\mathcal{A}$ (with the secret key corresponding to the public key A). Finally, it proves that B is the hash of some IP_C that the client knows. The latter part ensures that C has not unblinded the BLS signature (produced in the *periodic key generation phase* above) to a valid signature on some other random (unknown) message.

C then sends $\langle T_c, \Pi_{T_c} \rangle$ to the exit only for the *first* connection-throttling destination within the circuit.

On receiving the token T_c and the proof Π_{T_c}, the Tor exit verifies the proof to check if T_c is *well formed*, and terminates the circuit otherwise.

Stream token showing and verification phase. At this point C has obtained a blind signature from $\mathcal{A}$, has an active circuit through some exit, and has established a connection to some connection-throttling destination server-port combination, represented as $\mathcal{S}$. As defined earlier, let $n_{\mathcal{S}}$ be the maximum number of allowable unlinkable connections per client to destination $\mathcal{S}$. The client obtains the appropriate value of $n_{\mathcal{S}}$ for each destination $\mathcal{S}$ from the Tor consensus (see Section 9).

When the destination dynamically requests stream tokens, the Tor client C produces a deterministic stream token T_s and a zero-knowledge proof Π_{T_s} as follows:

1. *Token:* The token T_s is a deterministic function $f(\sigma_{\mathcal{A}}, cnt_{\mathcal{S}}, n_{\mathcal{S}}, n_{\mathcal{E}})$, where $cnt_{\mathcal{S}}$ is a counter that keeps track of the number of connections the Tor client C has made to the target destination $\mathcal{S}$ in a given ZXAD epoch and $n_{\mathcal{E}}$ is the ZXAD epoch number.

   Let $\langle h_1 = H_1(1, \mathcal{S}, n_{\mathcal{E}}) \ldots, h_{n_{\mathcal{S}}} = H_1(n_{\mathcal{S}}, \mathcal{S}, n_{\mathcal{E}}) \rangle$ be a public $n_{\mathcal{S}}$-value list of elements of $\mathbb{G}_1$ corresponding to $\mathcal{S}$. Note that both the client and the exit (or end server) can compute these values locally after obtaining the appropriate value of $n_{\mathcal{S}}$ from the Tor consensus. C computes the token T_s as follows:
$$T_s = e(h_\ell, \sigma_{\mathcal{A}}), 1 \le \ell \le n_{\mathcal{S}}$$
   where ℓ is the current value of the counter $cnt_{\mathcal{S}}$
2. *Zero-Knowledge Proof:* C constructs a non-interactive zero-knowledge proof Π_{T_s} (that T_s is *well formed*):
$$\Pi_{T_s} = PK\Big\{(\sigma_{\mathcal{A}}, \ell) : \big[T_s = e(h_\ell, \sigma_{\mathcal{A}}) \wedge 1 \le \ell \le n_{\mathcal{S}}\big] \bigwedge [\sigma_{\mathcal{A}} = \sigma_{\mathcal{A}}(T_c)]\Big\}$$
   where $\sigma_{\mathcal{A}}(T_c)$ denotes the $\sigma_{\mathcal{A}}$ in the circuit token T_c.

The proof has two parts: First it proves that the token T_s is of the form $e(h_\ell, \sigma_{\mathcal{A}})$, where h_ℓ is one of the n_S valid values (*i.e.*, $h_1 \ldots, h_{n_S}$), for some $\sigma_{\mathcal{A}}$. Next, it proves that the $\sigma_{\mathcal{A}}$ is the same value embedded in the circuit token. The latter part ensures that $\sigma_{\mathcal{A}}$ is a valid BLS signature on the hash of some IP_C that the client knows.

The Tor client C then sends $\langle T_s, \Pi_{T_s} \rangle$ to the exit. We observe that C can re-use the stream token and the proof as long as it makes connections (to the same destination-port combination) using the *same* circuit within a ZXAD epoch. Note that the $n_{\mathcal{E}}$ value at the circuit creation is used until the circuit expires.

On receiving $\langle T_s, \Pi_{T_s} \rangle$, the exit first checks it has not already seen T_s during the current (or the previous) epoch (*e.g.*, by keeping a hash table of stream tokens seen in the current and the previous epochs at any point in time). This ensures that the client does not "double spend" a token. The exit then verifies the proof to check if T_s is *well formed*, and takes any remedial action.

4.2 Exit abuse detection

We now describe how ZXAD can be used to combat exit-level and Tor-level attacks (defined in Section 3.1). Recall that circuit-level attacks can be rate limited without the use of ZXAD.

Exit-level attacks. The exits do not even require the cooperation of the destination server to combat exit-level attacks. ZXAD stream tokens provide a mechanism for individual Tor exits to rate limit the number of *unlinkable* connections to any target destination; *i.e.*, the number of *different circuits* containing streams to that destination. The exits can take any remedial action (plausibly decided by the maintainers of Tor) such as killing circuits that reuse tokens too often. We note that the exits can link circuits that reuse tokens to each other, but not back to a particular client.

Tor-level attacks. ZXAD stream tokens can further be forwarded by the exits to the destination servers to evade Tor-level attacks. As described in Section 3.4, the destinations can dynamically turn stream tokens on only when they are facing a large-volume attack from Tor (at large) or some Tor exit(s). The exits can then request clients to send a stream token, perform the token verification locally, and just forward well-formed stream tokens (or even just their hashes) to avoid burden on the destination servers (as shown in Figure 1). This provides much more fine-grained control to the destination servers — using the stream tokens, the servers can distinguish when one client (IP address) is making too many connections to them over Tor, even using multiple exits, and throttle them in the same way as they would throttle a non-Tor client making too many connections.

We discuss how the exits can forward the stream tokens and how the end servers can request exits to dynamically turn stream tokens on or off in Section 9.

4.3 Extension to t-out-of-n DirAuths

Using Shamir's secret-sharing scheme [47] (described in Section 2), ZXAD can be easily extended to Tor's existing t-out-of-n threshold DirAuths threat model. This guarantees that ZXAD is secure as long as a majority (5 out of 9) of the DirAuths are honest.

However, Shamir's secret-sharing scheme requires a central trusted dealer, which can securely generate and distribute secret shares to all DirAuths — Tor cannot afford such a high degree of trust in a single individual. Therefore, we use *additive* secret-sharing scheme [4] to first create a shared secret without a trusted dealer and then a share conversion scheme [18] to non-interactively create and update the Shamir secret shares.

Let the DirAuths be $\widehat{\mathcal{A}} = \{\mathcal{A}_1, \mathcal{A}_2, \ldots \mathcal{A}_n\}$. We consider subsets of $\widehat{\mathcal{A}}$ of size $n-(t-1)$. Let $\mathcal{P}_j, 1 \le j \le \binom{n}{t-1}$ be such subsets and let s_j be a random secret in $\mathbb{Z}_q$ for each $\mathcal{P}_j$. Then each $\mathcal{A}_i \in \mathcal{P}_j$ is given a copy of s_j by an arbitrary member of $\mathcal{P}_j$. Let $\alpha = \sum_{j=1}^{\binom{n}{t-1}} s_j$ be the joint secret key. Note that any t DirAuths between them hold all $\binom{n}{t-1}$ of the s_j values, and so can compute α, but any smaller set is missing at least one of the s_j, and so cannot compute α.

We now describe the share conversion procedure [18] to *non-interactively* convert the additive s_j shares of α into Shamir shares. For each $\mathcal{P}_j$, define the polynomial $g_j(x) = \prod_{i:\mathcal{A}_i \in \widehat{\mathcal{A}} \setminus \mathcal{P}_j} \frac{i-x}{i}$.

Note that for each $\mathcal{P}_j$ of size $n-(t-1)$, $g_j(x)$ is of degree $t-1$, satisfies $g_j(i) = 0$ for each $\mathcal{A}_i \in \widehat{\mathcal{A}} \setminus \mathcal{P}_j$, and $g_j(0) = 1$. Now define $f(x) = \sum_{\mathcal{P}_j} s_j \cdot g_j(x)$, which similarly is a degree $t-1$ polynomial. Each DirAuth $\mathcal{A}_i$ can compute $f(i)$ using their knowledge of s_j for each $\mathcal{P}_j$ that contains $\mathcal{A}_i$, but no other evaluation of f. Therefore, as $f(0) = \sum_{\mathcal{P}_j} s_j \cdot g_j(0) = \sum_{\mathcal{P}_j} s_j = \alpha$, each $f(i)$ is indeed a t-out-of-n Shamir secret share of α.

Importantly, the DirAuths need not communicate at all when updating this Shamir secret sharing of a random value in a forward-secret manner. We now describe in detail the changes to the initialization, rekeying, key publishing, and periodic key generation phases. There are no changes in any of the other phases.

- *Initialization phase.* Some DirAuth $\mathcal{A} \in \mathcal{P}_j$, $1 \le j \le \binom{n}{t-1}$ chooses $s_j \xleftarrow{R} \mathbb{Z}_q$ and sends it to all other DirAuths in the subset $\mathcal{P}_j \setminus \mathcal{A}$. Additionally, $\mathcal{A}$ adds a commitment to the Tor consensus [50], which can be verified by all other DirAuths that received the share s_j. The commitment is the hash of $\langle s_j, \mathcal{T}, n_w \rangle$, where $\mathcal{T}$ is the Tor shared randomness [32] and n_w is the ZXAD period number.

  Once all DirAuths have thus received their additive shares of α, they can independently compute their own Shamir shares α_i of α as described above, and $\langle \alpha_i, A_i = g_1^{\alpha_i} \rangle$ will

be used as their periodic key pair for the first ZXAD period. Finally, all DirAuths can publish their individual public keys A_i in the Tor consensus.

The DirAuths follow a similar procedure (to the one described above) to generate their long-term keypairs $\langle \rho_i, P_i = g_1^{\rho_i} \rangle$. A small change is that the DirAuths omit the period number in the commitments. As mentioned in Section 3.3 the long-term key is usually never changed (like the long-term identity keys in Tor [23]).

- *Long-term key generation phase.* C follows a similar procedure (described in the *periodic key generation phase* below) to receive the blind signature $\sigma_{\mathcal{P}}$ on its IP address. That is, C first chooses t of the DirAuths to contact, and performs the single-DirAuth long-term key generation phase protocol described in Section 4.1 with each of these t DirAuths, yielding t partial BLS signatures. C then combines these signatures to form $\sigma_{\mathcal{P}}$.

- *Rekeying phase.* Each DirAuth first increments $n_{\mathcal{W}}$ and uses a common Key Derivation Function (KDF) to independently convert each current share s_j to a new additive share $\widehat{s_j} = KDF(s_j, \mathcal{T}, n_{\mathcal{W}})$. The old s_j should be discarded for forward secrecy purposes. The DirAuths then proceed as above to independently compute their new $\langle \alpha_i, A_i = g_1^{\alpha_i} \rangle$ keypairs. Note that the rekeying phase is completely noninteractive.

- *Key publishing phase.* At the beginning of every ZXAD period, when the Tor DirAuths generate the first hourly consensus, they can compute and publish their individual public keys A_i that they will be using in the current and the next periods. At the beginning of every period, all exits and clients can update their view of the A_i values (for the current and the next period respectively) from the Tor consensus. Additionally, the clients compute the joint public key (for the next period) $A = \prod_{i=1}^{t} A_i^{\lambda_i}$, where λ_i is the Lagrange coefficient for interpolating on the set $\{1, 2 \ldots, t\}$.

- *Periodic key generation phase.* To receive its blind signature, C specifies which keypair the DirAuths need to use (*i.e.*, the current or the next period's) and chooses t of the DirAuths to contact; say $\{\mathcal{A}_i\}_{i \in V}$, where V is a subset of $\{1, \ldots, n\}$ of size t. C then performs the single-DirAuth periodic key generation phase protocol described in Section 4.1 with each of these t DirAuths, yielding t partial blind signatures $\langle \sigma_i \rangle_{i \in V}$ with the specified period's secret key. C then combines these signatures to form $\widetilde{\sigma}_{\mathcal{A}} = \prod_{i \in V} \sigma_i^{\lambda_i}$, where the λ_i are the Lagrange coefficients for interpolating over the set of indices V. Finally, C uses the specified period's public key to verifies if $\widetilde{\sigma}_{\mathcal{A}}$ is a valid signature by the DirAuths and unblinds it.

5 Security

ZXAD is a zero-knowledge based protocol that helps Tor exits to detect large-volume traffic to a target server (by a single Tor client), without revealing any information about the client. It uses Tor's existing threat model; *i.e.*, the anonymity of a Tor client is compromised if a majority of the Tor directory authorities are compromised.

The security of ZXAD relies on the security of (i) the blind signature transfer; (ii) the BLS signature used by the DirAuths to issue the long-term and periodic client keys; (iii) the DirAuths' threshold key generation and non-interactive rekeying protocol (see Section 4.3); (iv) the zero-knowledge proof (ZKP) used to prove that the circuit token is well formed; and (v) the ZKP used to prove that the stream token is well formed.

The security of the (blind) BLS signature scheme [7] and the share conversion scheme [18] that we adapt for ZXAD imply the security for steps (i), (ii), and (iii) above. Therefore, we focus on the security of our ZKPs (*i.e.*, (iv) and (v) above) from here on.

ZXAD uses two ZKPs as sub-protocols: (i) the Discrete Log Equality (*DLE*) and (ii) the Discrete Log Product Equality (*DLEP*). *DLE* is the standard Chaum-Pedersen proof of equality of discrete logs [17], while *DLEP* is a Generalized Schnorr Proof [12] of a discrete log product. For reference, we define these ZKPs and prove their security in Appendix A.

5.1 Circuit Token Zero-knowledge Proof

We now describe and prove correct a zero-knowledge proof that proves that token T_c is well formed:

(i) the client knows some r_2 such that $g_2'' = g_2^{r_2}$ and $\sigma_{\mathcal{A}}'' = \sigma_{\mathcal{A}}^{r_2}$, for some $\sigma_{\mathcal{A}}$; (ii) $\sigma_{\mathcal{A}}$ is a valid signature on some B by the DirAuth $\mathcal{A}$ (with the secret key corresponding to the public key A); and (iii) B is the hash of some IP_C that the client knows.

To prove that $\sigma_{\mathcal{A}}$ is a valid signature, we first randomize B by setting $B'' = B^{r_2}$. As B'' is uniform in $\mathbb{G}_2$, we reveal its value rather than proving knowledge of it. Next, we formulate the following proof statements (the secret witnesses are underlined for clarity).

$$\Pi_{T_c} = PK\Big\{(\underline{r_2}, \underline{IP_C}) :$$
$$\hat{s}_{T_{c1}} : \; e(g_1, \sigma_{\mathcal{A}}'') = e(A, B'')$$
$$\hat{s}_{T_{c2}} : \; [g_2'' = g_2^{\underline{r_2}}] \wedge [B'' = H_2(\underline{IP_C})^{\underline{r_2}}]\Big\}$$

We observe that statement $\hat{s}_{T_{c2}}$ proves knowledge of a pre-image under H_2. This is hard to prove using a Σ-protocol. Therefore, we use a zkSNARK [5] instead.

With the zkSNARK proving knowledge of r_2 and IP_C, statement $s_{T_{c1}}$ then shows that $\sigma_{\mathcal{A}}''$ can be unblinded to some $\sigma_{\mathcal{A}}$ that is a valid BLS signature on $B = H_2(IP_C)$. Also, $\hat{s}_{T_{c1}}$ does not involve any secret terms and hence can be easily verified by the verifier (*i.e.*, DirAuth $\mathcal{A}$).

Recall (from Section 4.1) that the circuit tokens can be computed offline by choosing $r_2 \xleftarrow{R} \mathbb{Z}_q^*$. Further, we note that the private inputs to the zkSNARK are just the client's IP address and r_2. Therefore the client can compute the zkSNARK proofs also offline along with the circuit tokens. This way the clients' most expensive step in ZXAD can be performed completely offline.

Let Π'_{T_c} be the zkSNARK proof. The client sends $\sigma''_{\mathcal{A}}$ and Π'_{T_c} (which contains g_2'' and B'') to the Tor exit. g_1, g_2, A are public and known to both client and exit. The exit verifies the zkSNARK proof and statement $\hat{s}_{T_c 1}$.

It is easy to check that our zero-knowledge proof is *complete*. We prove the *soundness* and the *zero-knowledgeness* in Appendices B.1 and B.2 respectively.

5.2 Stream Token Proof Σ-Protocol

We now describe and prove correct a Σ-protocol that proves the stream token T_s is well formed: (i) $T_s = e(h_\ell, \sigma_{\mathcal{A}})$, where h_ℓ is one of the n_S valid values $h_1, h_2 \ldots, h_{n_S}$ (defined in Section 4.1), for some $\sigma_{\mathcal{A}}$ and (ii) $\sigma_{\mathcal{A}}$ is the same value embedded in the circuit token.

To prove that token T_s is well formed, we formulate a Σ-protocol that proves these statements (the secret witnesses are underlined for clarity). Note that $s_{T_s 2}$ shows that the $\sigma_{\mathcal{A}}$ is the same value committed to as $\langle g_2'' = g_2^{r_2}, \sigma''_{\mathcal{A}} = \sigma_{\mathcal{A}}^{r_2} \rangle$ in the circuit token T_c.

$$s_{T_s 1} : \bigvee_{i=1}^{n_S} \; T_s = e(h_i, \underline{\sigma_{\mathcal{A}}})$$

$$s_{T_s 2} : \; [g_2'' = g_2^{\underline{r_2}}] \wedge [\sigma''_{\mathcal{A}} = \underline{\sigma_{\mathcal{A}}}^{r_2}]$$

We observe that statement $s_{T_s 1}$ is an OR-proof and hence the proof size grows linearly with n_S, and could potentially be expensive if the prover or verifier had to compute n_S pairings.

Therefore to prove $s_{T_s 1}$ (without n_S pairings), we choose $r_1 \xleftarrow{R} \mathbb{Z}_q$ and compute public component $Y_1' = Y_1^{r_1} \cdot h_\ell$, where ℓ is the correct value of i in $s_{T_s 1}$ such that $T_s = e(h_\ell, \sigma_{\mathcal{A}})$. Additionally, to prove knowledge of r_1, we compute another public component $g_1' = g_1^{r_1}$.

Now we rewrite the proof statements as follows (the secret witnesses are underlined for clarity):

$$\Pi_{T_s} = PK\Big\{(\underline{\ell}, \underline{r_1}, \underline{r_2}) :$$

$$\hat{s}_{T_s 1} : \bigvee_{i=1}^{n_S} [i = \underline{\ell}] \; DLE_{\underline{r_1}}\big[g_1, g_1', Y_1, Y_1' \cdot h_i^{-1}\big]$$

$$\hat{s}_{T_s 2} : \; DLEP_{\underline{r_1}, \underline{r_2}}\big[g_1, g_1', g_2, g_2'', e(Y_1, \sigma''_{\mathcal{A}}), T_s, e(Y_1', \sigma''_{\mathcal{A}})\big]\Big\}$$

We use the Chaum-Pedersen Σ-protocol to prove knowledge of r_1, a Borromean ring OR proof [38] to prove knowledge of ℓ, and our DLEP Σ-protocol to prove knowledge of r_2 and that the token $T_s = e(h_\ell, \sigma_{\mathcal{A}})$ for some $\sigma_{\mathcal{A}}$ such that $\sigma''_{\mathcal{A}} = \sigma_{\mathcal{A}}^{r_2}$.

The client sends $\langle T_s, \Pi_{T_s} \rangle$ to the Tor exit. $g_1, g_2, Y_1, h_1 \ldots, h_{n_S}$ are public and are known (or can be computed) by both the client and the exit.

We summarize the complete Σ-protocol in Appendix C. It is easy to check that our ZKP is *complete*. We leave the *soundness* and the *zero-knowledgeness* proofs to Appendices C.1 and C.2 respectively.

Optimization. To improve the performance of ZXAD, computing the product of multiple pairings (in Sections 4.1, 5.1, and 5.2) can be optimized by computing the product of the Miller loops [40], followed by a single final exponentiation. This makes the cost of computing a *batch pairing* (of three pairings) roughly the same as that of two individual pairings.

6 Implementation

We built two proof-of-concept implementations for our zero-knowledge proofs: (i) in C++ using the libsnark [36] library (ii) in Go using the Kyber [20] cryptographic library.

We implemented the complete ZXAD protocol (to test its correctness) over the MNT curve [41] of embedding degree 4 using libsnark. However, libsnark does not have well-optimized implementations of group operations (see Section 7 for timing comparisons). Therefore, we also implemented a faster Go version using the Kyber [20] library (to evaluate the performance). Since the Kyber library does not support zkSNARKs or the MNT curves, we implemented all of our zero-knowledge proofs (except the zkSNARK) over the 256-bit Barreto-Naehrig curve [42].

Our implementations are available for download at https://git-crysp.uwaterloo.ca/iang/zxad.

7 Evaluation

To evaluate the performance of ZXAD, we first tested the end-to-end libsnark implementation for correctness. Next to evaluate the performance of ZXAD, we performed a series of micro-benchmarks on both the libsnark and the Kyber implementations. All experiments were run using a single thread (since Tor mainly uses a single thread [37]) on a 4.00 GHz i7-6700K desktop machine running Ubuntu 16.04.

Experimental setup. We evaluate ZXAD mainly by considering the load placed on the DirAuths (which verify the long-term key and issue the periodic key) and the exits (which verify the circuit and stream token proofs). To be practical, ZXAD should incur low overheads for both the DirAuths and the exits. We observe that the bulk of ZXAD's overhead is the blind signature transfer and the zero-knowledge proofs. We therefore measure the load placed on the DirAuths and the exits in terms of the computation (*i.e.*, the verifying times) and the communication costs (*i.e.*, sizes) for these operations.

Table 1: The mean and standard deviation over 2500 runs of different operations.

Library	Operation	Offline Execution Time (ms)	Verifying Time (ms)	Size (bytes)
Kyber	Blind signature transfer	0.83 ± 0.04	1.82 ± 0.04	256
	Circuit token generation	0.44 ± 0.07	1.81 ± 0.04	128
libsnark	Blind signature transfer	2.91 ± 0.08	2.60 ± 0.03	800
	Circuit token generation	1.46 ± 0.06	2.62 ± 0.05	160
	zkSNARK proof	3270 ± 20	8.4 ± 0.1	169

Additionally, we also measure the load placed on the clients in terms of the execution times for these operations.

We measure (a) the execution and verification times and (b) the size for the blind signature transfer and the two ZXAD zero-knowledge proofs (*i.e.*, the circuit and stream token proofs). We observe that the performance of our circuit token proof is independent of the destination visited (*i.e.*, $\mathcal{S}$) and the current connection count (*i.e.*, ℓ) to that destination. However, the OR-proof (in the stream token proof) depends on n_s, the threshold number of unlinkable connections (circuits containing streams) to the destination $\mathcal{S}$ in a given epoch. Therefore, to explore how n_s affects the performance, we consider a Tor client that connects to a regular connection-throttling destination and vary n_s from 1 to 25 in our stream token proof experiments.

For the execution and verification times, we repeat each experiment 2500 times and report the mean over the 2500 iterations with their standard deviations (see Table 1). Note that the blind signature transfer, the circuit token, and the zkSNARK proof can be computed *offline* (see Sections 4.1 and 5.1) and therefore all the execution times reported in Table 1 are *offline* execution times.

For the size experiments, we run each experiment once and report the results in Table 1 (as the communication cost does not vary for every run).

Finally, for the stream token proof experiments we measure both proving and verifying times and the proof size 100 times for every value of n_s from 1 to 25 and plot the results in Figures 3 and 4. We use the results from the Kyber experiments for our analysis (unless otherwise explicitly stated) as they are significantly faster than the libsnark results.

Load on the DirAuths. Recall (from Section 4.1) that the DirAuths: (i) issue a long-term key to new Tor clients, (ii) verify before the blind signature transfer, and (iii) issue a periodic key to every Tor client. As mentioned before, to be practical ZXAD should incur low overheads for the DirAuths, which may have a large volume of clients connecting to them.

To evaluate the suitability of ZXAD for Tor, we derive our "ground truth" — the number of new clients connecting to Tor in a week — using data from the Tor Metrics Portal [52]. Tor reports ~ 1.9 million daily users [52] in November 2020. Assuming the worst (and unlikely) case that all daily users connecting to Tor in a week are new and unique (*i.e.*, require both long-term and periodic keys), the DirAuths would get around $1.9 \times 7 \approx 13.4$ million long-term and periodic key requests in a week.

We now consider the computation overhead for the DirAuths while issuing the long-term and the periodic keys, each of which involves issuing a BLS signature. We find that a BLS signature computation takes 0.40 ± 0.02 ms. Next, we consider the computation overhead for the DirAuths while verifying before the blind signature transfer. From Table 1, we observe that the verification time is 1.82 ms. Therefore the total overhead on the DirAuths is $2 \times 0.4 + 1.82 \approx 2.62$ ms. That is, the DirAuths can handle up to ~ 381 clients per second. Moreover, since the DirAuths are usually multiple-core machines, they can easily verify the proofs in parallel on the spare cores. Therefore for the default ZXAD period of one week, even with a single spare core, the DirAuths can verify up to $381 \times 3600 \times 24 \times 7 \approx 230.4$ million clients. In other words, the DirAuths can easily handle far more than the expected 13.4 million long-term and periodic key requests in a week.

Next we consider the communication overhead for the DirAuths. We find that the size of a BLS signature is 128 bytes. From Table 1, we observe that the blind signature transfer communication cost is 256 bytes. So, the overall computational overhead on the DirAuths is $2 \times 128 + 256 \approx 512$ bytes. Therefore, even for handling 230.4 million clients in a week (*i.e.*, 381 clients per second), the DirAuths would just require a low bandwidth of $512 \times 381 \times 10^{-3} \approx 196$ KB/sec.

Load on the exits. Recall (from Section 4.1) that a Tor client generates: (i) the circuit token proof once per circuit (for the *first* connection-throttling destination) and (ii) the stream token proof once per circuit for every unique destination. As mentioned before, to be practical ZXAD should incur low overheads for the exits, which may have numerous circuits created through them. Note that we are interested in the number of circuits (and not streams) created per exit per epoch, as ZXAD rate-limits clients based on n_s, the maximum number of allowable *unlinkable* connections (or circuits containing streams) to the destination $\mathcal{S}$ every epoch. Hence to evaluate the suitability of ZXAD for Tor, here we derive our "ground truth" — the maximum number of circuits per exit per epoch — using empirical values modeled from the Tor network [35] and data from the Tor Metrics Portal [52]. Komlo et al. [35] report that on an average, 8.9 circuits are created every hour per client. Tor reports ~ 1.9 million daily users [52] in November 2020, so the total number of circuits created across Tor every hour is $8.9 \times 1,900,000 \approx 17,000,000$ circuits. That is, in every ZXAD epoch $17,000,000/6 \approx 2,850,000$ circuits are being created across all exits. From the Tor Metrics Portal [52], the current maximum weighted exit has an exit weight equaling $\sim 0.45\%$ of the total available exit weight in Tor. Therefore, the maximum number of circuits created through a single Tor exit every ZXAD epoch is $0.0045 \times 2,850,000 \approx 13,000$ circuits.

Since the Kyber library does not support zkSNARKs, we use the libsnark results just for the zkSNARK analysis.

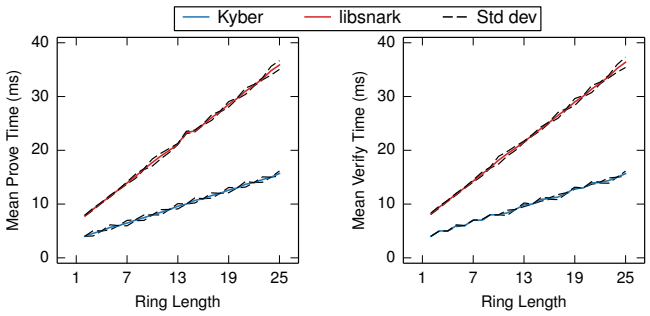

Figure 3: The mean of 100 runs of prove (*left*) and verify (*right*) times for our stream token proof Σ-protocol as a function of n_S, the maximum number of unlinkable connections allowed to a destination $\mathcal{S}$. The dashed lines represent the standard deviation.

Though the zkSNARK proof is implemented over a different curve (*i.e.*, the MNT4 curve), combining the results would give us an approximate measure of the overheads for verifying a circuit proof. This is because all zkSNARKs are fast to verify (just a few *milliseconds*) and result in very small proofs (less than 500 bytes).

We first focus on the overall computational overhead for the exits. First, we observe that the time taken for verifying the circuit token and the zkSNARK proof is 1.81 ms and 8.4 ms respectively. Therefore the exit takes a total of $1.81 + 8.4 = 10.21$ ms to verify the circuit token and the proof. Next, we focus on the computation overhead for verifying a stream token proof, which also involves computation of the n_S-value list (*i.e.*, hashing to $\mathbb{G}_1$, n_S times). We find that for a reasonable value of $n_S = 10$, the hashing to $\mathbb{G}_1$ (which is a linear function of n_S) and the stream token proof verification take 0.55 ± 0.01 ms and 8 ms (from Figure 3) respectively. That is, overall the busiest exit takes $(10.21 + 0.55 + 8) \times 13{,}000 \approx 243{,}880$ ms or 4.1 minutes per epoch for verification. However, this 4.1 minutes overhead is only in the worst (and unlikely) case where every circuit through the busiest Tor exit contains a stream to a connection-throttling destination under an attack. Note that for subsequent streams connecting to new connection-throttling destinations (under attack) within the same circuit, the exit needs to verify only the stream token. This reduces the verification time to almost half for subsequent destinations.

Moreover, we observe that the cryptographic verification of the circuit (and the stream) tokens is *embarrassingly parallel*; *i.e.*, the most overloaded (or the high-bandwidth) exits can easily verify multiple tokens in parallel on a multi-core machine. Therefore, even an eight-core processor can reduce exit verification time further down to ~ 31 s per epoch even in the worst case.

We now consider the overall communication overhead for the exits. First, we observe that (from Table 1) the circuit token and the zkSNARK proof sizes are 128 and 169 bytes respectively. Next from Figure 4, we observe that, up to a reasonable value of $n_S = 10$, the stream token proof size is

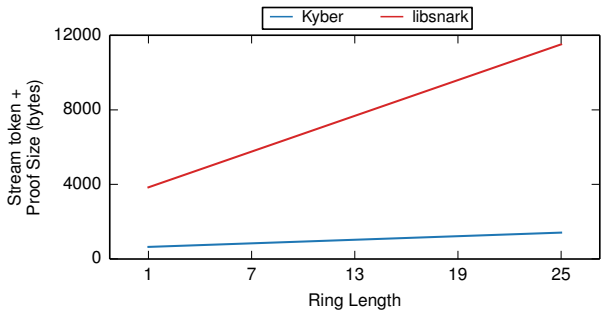

Figure 4: The stream token and proof (*i.e.*, T_s and Π_{T_s}) size for our stream token proof Σ-protocol as a function of n_S, the maximum number of unlinkable connections allowed to a destination $\mathcal{S}$.

928 bytes. Therefore, the overall communication cost incurred by the exits is $128 + 169 + 928 = 389$ bytes per circuit. That is, for handling 13,000 circuits per epoch (in the worst case), the busiest exit would just require a low bandwidth of $(389 \times 13{,}000 \times 10^{-3})/(10 \times 60) \approx 8.5$ KB/sec.

Load on the clients. Recall (from Section 4.1) that a Tor client computes: (i) the computation for the blind signature transfer once every period; (ii) the circuit token proof once per circuit (for the *first* connection-throttling destination) and (iii) the stream token proof once per circuit for every unique destination.

As already mentioned (in Sections 4.1 and 5.1), (i) and (ii) above can be computed offline anytime after the client gets its long-term and periodic key respectively. Therefore, the client only creates the stream token proof online (which also involves computation of the n_S-value list). As already mentioned for a reasonable value of $n_S = 10$, the hashing to $\mathbb{G}_1$ takes 0.55 ± 0.01 ms. From Figure 3, we observe that the proving time for the stream token proof is 8 ms (for $n_S = 10$). Therefore, the overall online overhead of the client is 8.55 ms. That is, the client would just experience a latency of 8.55 ms every time it accesses a new connection-throttling website (under attack) via the Tor Browser and a regular load time for all subsequent accesses in a given epoch. Therefore, the 8.55 ms overhead is negligible for the client.

8 Related Work

Attacks stemming from Tor can be caused by malicious exits themselves or by benign exits that are being abused by malicious users.

Malicious exits. Prior research works [16; 39; 59] have found evidence of malicious behavior such as traffic snooping, SSL stripping, *etc.* by Tor exit relays. To mitigate these attacks, the Tor project actively scans for "bad" exit relays using tools like exitmap [58], sybilhunter [57], and torscanner [1]. Moreover, Tor users can also report suspicious activities performed by misconfigured or malicious exits [56]. Once a suspected activity is reported, it is reproduced and

verified. Then, based on the severity of the attack, the exit is assigned one of the three flags — BadExit, Invalid, or Reject — so that clients will no longer select them as the last hop (or for any hop).

Exit abuse. Tor currently does not have any built-in mechanisms to prevent benign exits being abused by malicious users. There has been a considerable line of research [8; 9; 10; 28; 29; 31; 54; 55] in anonymous blacklisting and revocation systems in the past. However, as Henry and Goldberg [27] mention most of these systems either offer weaker privacy guarantees, such as *linkable pseudonymity*, or leverage (semi-)trusted third parties to provide anonymity, or incur high computational overhead for service providers and users of the system.

Differential treatment to Tor users. A recent study by Khattak et al. [34] showed that website operators have started providing second-class treatment to all Tor users, to mitigate the attacks stemming from Tor. Tor users now often face CAPTCHAs or even outright blocking. Their study showed that 3.67% of Alexa top 1000 sites were blocking Tor users and many publicly available Tor blacklists [3; 15] have evolved.

Singh et al. [48] characterized the nature of undesired traffic originating from Tor by considering e-mail contents sent to exits, blacklisting of Tor relays, and the server response to Tor traffic. They found that 7% of IP blacklists list exit IPs immediately after they were listed in the consensus. Moreover they found that a majority of the attacks stemming from Tor were large-volume ones, such as DDoS, port scanning, *etc.* suggesting possibilities of privacy-preserving detection and mitigation.

Related cryptographic protocols. Camenisch et al. [13] propose a *n*-times anonymous authentication system that relies on a Public Key Infrastructure (PKI) trust setting. In their scheme, each user generates their own key pair and gets anonymously authenticated from a single credential issuer using CL signatures [11] (which are far slower than the BLS signatures [7] used in ZXAD). Our solution uses a completely different approach that yields a practical, efficient, and more suitable solution for the Tor network. In our approach, we use a distributed issuer with malicious minority setting (just like the DirAuths in Tor) so that the users can be individually authenticated by the issuers. We also provide a method to non-interactively generate and update (in a forward-secret manner) a joint secret key among the issuers (*i.e.*, the DirAuths) and evaluate the suitability of ZXAD for Tor.

Existing solutions. Privacy Pass [19] is a zero-knowledge based solution to prevent users (of Tor mainly) from being victims of a disproportionate amount of internet challenges such as CAPTCHAs. It grants users 30 anonymous tokens for every CAPTCHA they solve; these tokens may be used later in an unlinkable manner to avoid future CAPTCHAs. However, the computational asymmetry [27] may allow some users (especially the ones computationally capable to execute a large-volume attack) to obtain more tokens than others. Therefore, Privacy Pass is more a CAPTCHA avoidance solution, than an exit abuse solution [2].

Opportunistic onions [46], introduced by Cloudflare, uses Tor's onion service protocol to monitor and limit individual circuits — while a destination server views the same IP address (*i.e.*, the Tor exit IP) for each individual Tor client connection or circuit, an onion service views a unique ephemeral circuit ID number. Opportunistic onions uses this ephemeral ID to rate limit the circuit. Malicious users may still repeat the onion service protocol and establish a fresh circuit, but doing so involves repeating the costly Tor rendezvous protocol.

9 Discussion and Limitations

In this section, we discuss practical aspects of deploying ZXAD, and some of its limitations.

Choice of n_s values. An important question for ZXAD is selecting an appropriate value for n_s, the maximum number of allowable unlinkable connections (on *different* circuits) per client to a given destination $\mathcal{S}$, so that the abuse detection is not triggered in the normal course of browsing. We come up with some reasonable n_s values for different types of destinations based on how Tor operates: (i) unlimited for very popular destinations (such as Google ads, analytics, *etc.*) and Alexa top 1000 sites that are likely to appear in multiple tabs (recall each first-party tab gets its own circuit in Tor Browser); (ii) 10 or a moderate value for third-party services such as OAuth that one expects to see embedded in multiple first-party tabs; and (iii) 1 or 2 for other sites.

At the beginning of every day, the DirAuths can add the hash of destinations in category (i) and (ii) above to the Tor consensus, and all clients and exits can update their view of the n_s values. We suggest updating the n_s values once per day, since updating even the hash of 1000 or so destinations every hour can be quite tedious.

Sending stream tokens to destination servers. To combat Tor-level attacks as described in Section 4.2 (with the cooperation of the destination server), we suggest sending a hash of the ZXAD stream token along with the TCP connection from the exit to the server, perhaps by embedding it in a TCP option [53] or by having a separate application-level service for sending (and receiving) ZXAD stream tokens. The TCP option [53] solution is somewhat similar to Cloudflare's [46] approach of encoding the circuit ID as an IPv6 address and using the Proxy Protocol header [49] for sending it to the destination server. The server would then check if it had seen the stream token hash before (from *any* exit), closing the connection if it had.

A similar approach can be used by the destination servers to dynamically turn tokens on or off (by encoding the operation as a single bit).

zkSNARK deployment. The deployment of the zk-

SNARK version requires a Common Reference String (CRS) to generate the initialization parameters. Precautions must be taken to destroy this initial secret, as otherwise anyone who has access to the secret can generate false proofs. We envision that the maintainers of the Tor Project can follow similar steps followed by other popular zkSNARK based systems such as Zcash [30], but the contributors can simply be the DirAuths, a majority of which are assumed honest already.

Denial-of-Service attacks. A malicious client can disrupt ZXAD (*i.e.*, cause denial-of-service) by submitting malformed stream and circuit tokens or proofs. ZXAD is not immune to this type of DoS attack. However, the damage done can be minimized (for malformed stream tokens or proofs) by rate-limiting the number of streams per circuit at the exits. Malicious clients can still DoS ZXAD by submitting malformed circuit or stream tokens and proofs through different circuits. In this case, the exits can first verify the SNARK and the DLEP proofs (defined in Sections 5.1 and 5.2) which are significantly smaller in size, and reject all proofs that do not verify. Then for the remaining "almost-verifiable" responses, it can verify the complete circuit (or stream) token proof. We believe that the latter case will not be that common, since the client needs to spend ~ 9 ms (for $n_s = 10$) of online computation per stream token for generating such almost-verifiable proofs. This is, to some degree, similar to the opportunistic onions solution [46], wherein malicious clients creating a fresh circuit have to repeat the time-consuming "rendezvous protocol" over and over again.

Malicious clients can also try to disrupt ZXAD by submitting far too many long-term or periodic key requests to the DirAuths. The DirAuths can limit long-term key requests by dynamically requesting a client proof of work (*e.g.*, a computational puzzle) [27]. Note that, since the long-term key request is performed infrequently, this does not affect honest clients much. The DirAuths can rate-limit periodic key requests (without even turning on proof of work) — honest clients that had obtained their periodic key in advance (in the previous ZXAD period) are not affected by this in any way. All other clients can still access destinations (that are not under attack) through Tor, as today, and resend a request for the periodic key later.

10 Conclusion

We present ZXAD, a zero-knowledge based exit abuse detection system for Tor, that detects large-volume attacks (*e.g.*, DoS attacks) by a single Tor client in a privacy-preserving way. ZXAD does not reveal any information other than the fact that some client is making numerous connections to a target destination. Unlike existing work, ZXAD has wide applicability — rather than just relying on the high computational cost for performing a large-volume attack, ZXAD allows a threshold to be set (per destination server) for the number of per-client connections allowed through Tor in a given epoch and helps to detect Tor users that exceed this limit. We formally prove that ZXAD provides strong privacy guarantees as long as a majority of the DirAuths are honest.

Additionally, we propose a t-out-of-n threshold DirAuth key generation protocol for ZXAD, which allows DirAuths to rekey a Shamir-shared private key in a forward-secret manner without any communication between the DirAuths.

We demonstrate using proof-of-concept implementations that on an average ZXAD incurs ~ 8.55 ms (on a single core) of client-side computation, 31 s (using eight cores) of exit-side computation per 10-minute epoch for the busiest exit in the worst case, and an exit-side bandwidth of at most 8.5 KB/sec, making it practical for Tor. We envision that ZXAD, if deployed in Tor, could reduce high-bandwidth exit abuse to a great extent and in turn encourage more volunteers to run exit relays.

Acknowledgments

This research was undertaken, in part, thanks to funding from the Canada Research Chairs program.

References

[1] torscanner: A console application to track bad exit nodes on Tor. https://github.com/torscanner/torScanner, 2013. Accessed Dec 2020.

[2] Review Cloudflare's Official "Privacy Pass" addon to evaluate inclusion in Tor Browser. https://trac.torproject.org/projects/tor/ticket/24321, 2017.

[3] Daniel Austin. TOR Node List. https://www.dan.me.uk/tornodes, 2020. Accessed Dec 2020.

[4] Josh Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In *Conference on the Theory and Application of Cryptography*, pages 27–35. Springer, 1988.

[5] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 326–349, 2012.

[6] Dan Boneh. The decision diffie-hellman problem. In *Proceedings of the Third Algorithmic Number Theory Symposium*, pages 48–63, 1998.

[7] 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.

[8] Stefan Brands, Liesje Demuynck, and Bart De Decker. A practical system for globally revoking the unlinkable pseudonyms of unknown users. In *Australasian Conference on Information Security and Privacy*, pages 400–415. Springer, 2007.

[9] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID: A direct anonymous attestation scheme with enhanced revocation capabilities. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 21–30, 2007.

[10] Jan Camenisch and Anna Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *International conference on the theory and applications of cryptographic techniques*, pages 93–118. Springer, 2001.

[11] Jan Camenisch and Anna Lysyanskaya. A signature scheme with efficient protocols. In *International Conference on Security in Communication Networks*, pages 268–289. Springer, 2002.

[12] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In *Annual International Cryptology Conference*, pages 410–424. Springer, 1997.

[13] Jan Camenisch, Susan Hohenberger, Markulf Kohlweiss, Anna Lysyanskaya, and Mira Meyerovich. How to Win the Clone Wars: Efficient Periodic n-Times Anonymous Authentication. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 201–210, 2006.

[14] Jan Camenisch, Aggelos Kiayias, and Moti Yung. On the Portability of Generalized Schnorr Proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 425–442. Springer, 2009.

[15] Inc. CGP Holdings. DNSBL.info: Spam Database Lookup. https://www.dnsbl.info/dnsbl-details.php?dnsbl=exitnodes.tor.dnsbl.sectoor.de, 2020. Accessed Dec 2020.

[16] Sambuddho Chakravarty, Georgios Portokalidis, Michalis Polychronakis, and Angelos D Keromytis. Detecting Traffic Snooping in Tor using Decoys. In *Recent Advances in Intrusion Detection (RAID)*, 2011.

[17] David Chaum and Torben P. Pedersen. Wallet Databases with Observers. In *Advances in Cryptology (CRYPTO '92)*, 1992.

[18] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography Conference*, pages 342–362. Springer, 2005.

[19] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy Pass: Bypassing Internet Challenges Anonymously. *Proceedings on Privacy Enhancing Technologies*, 2018(3):164–180, 2018.

[20] Decentralized and Distributed Systems Lab. kyber: Dedis advanced crypto library for go. https://godoc.org/go.dedis.ch/kyber, 2020.

[21] Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, and Bryan Parno. Cinderella: Turning shabby x. 509 certificates into elegant anonymous credentials with the magic of verifiable computation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 235–254. IEEE, 2016.

[22] Roger Dingledine and Nick Mathewson. Tor Protocol Specification. https://spec.torproject.org/tor-spec, 2020.

[23] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, pages 303–320, 2004.

[24] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology CRYPTO '86*, pages 186–194. Springer, 1986.

[25] Steven D Galbraith, Kenneth G Paterson, and Nigel P Smart. Pairings for cryptographers. *Discrete Applied Mathematics*, 156(16):3113–3121, 2008.

[26] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1):186–208, 1989.

[27] Ryan Henry and Ian Goldberg. Formalizing anonymous blacklisting systems. In *2011 IEEE Symposium on Security and Privacy*, pages 81–95. IEEE, 2011.

[28] Ryan Henry, Kevin Henry, and Ian Goldberg. Making a Nymbler Nymble using VERBS. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 111–129. Springer, 2010.

[29] Jason E Holt and Kent E Seamons. Nym: Practical pseudonymity for anonymous networks. *Internet Security Research Lab Technical Report*, 4:1–12, 2006.

[30] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. *GitHub: San Francisco, CA, USA*, 2016.

[31] Peter C Johnson, Apu Kapadia, Patrick P Tsang, and Sean W Smith. Nymble: Anonymous IP-address blocking. In *International Workshop on Privacy Enhancing Technologies*, pages 113–133. Springer, 2007.

[32] George Kadianakis. Mission: Montreal! (Building the Next Generation of Onion Services). https://blog.torproject.org/mission-montreal-building-next-generation-onion-services, 2016.

[33] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. Limits of anonymity in open environments. In *International Workshop on Information Hiding*, pages 53–69. Springer, 2002.

[34] Sheharbano Khattak, David Fifield, Sadia Afroz, Mobin Javed, Srikanth Sundaresan, Damon McCoy, Vern Paxson, and Steven J Murdoch. Do You See What I See? Differential Treatment of Anonymous Users. In *Network and Distributed Systems Security Symposium*. The Internet Society, 2016.

[35] Chelsea Komlo, Nick Mathewson, and Ian Goldberg. Walking Onions: Scaling Anonymity Networks while Protecting Users. In *29th USENIX Security Symposium*, 2020.

[36] SCIPR Lab. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark, 2020.

[37] Nick Mathewson. Threads in Tor. https://people.torproject.org/~nickm/tor-auto/internal/01f-threads.html, 2015.

[38] Gregory Maxwell and Andrew Poelstra. Borromean ring signatures, 2015.

[39] Damon McCoy, Kevin Bauer, Dirk Grunwald, Tadayoshi Kohno, and Douglas Sicker. Shining Light in Dark Places: Understanding the Tor Network. In *Privacy Enhancing Technologies Symposium*, 2008.

[40] Victor S. Miller. Short programs for functions on curves. IBM Thomas J. Watson Research Center, 1986.

[41] Atsuko Miyaji, Masaki Nakabayashi, and Shunzou Takano. New explicit conditions of elliptic curve traces for FR-reduction. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 84(5):1234–1243, 2001.

[42] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *International Conference on Cryptology and Information Security in Latin America*, pages 109–123. Springer, 2010.

[43] The Open Information Security Foundation. Suricata: An open source network threat detection engine. https://suricata-ids.org/, 2020. Accessed Dec 2020.

[44] Matthew Prince. The Trouble with Tor. Cloudflare Blog Post, 2016. https://blog.cloudflare.com/the-trouble-with-tor/.

[45] John Roberts. Control your traffic at the edge with Cloudflare. https://blog.cloudflare.com/cloudflare-traffic/, 2016.

[46] Mahrud Sayrafi. Introducing the Cloudflare Onion Service. https://blog.cloudflare.com/cloudflare-onion-service/, 2018.

[47] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[48] Rachee Singh, Rishab Nithyanand, Sadia Afroz, Paul Pearce, Michael Carl Tschantz, Phillipa Gill, and Vern Paxson. Characterizing the nature and dynamics of Tor exit blocking. In *26th USENIX Security Symposium*, pages 325–341, 2017.

[49] Willy Tarreau. The PROXY protocol. https://www.haproxy.org/download/1.8/doc/proxy-protocol.txt, 2020.

[50] Inc. Tor Project. Tor directory protocol, version 3. https://gitweb.torproject.org/torspec.git/plain/dir-spec.txt, 2020.

[51] The Tor Project. Who uses Tor? https://www.torproject.org/about/torusers.html, 2019.

[52] The Tor Project. Tor Metrics Portal. https://metrics.torproject.org/, 2020.

[53] Viet-Hoang Tran and Olivier Bonaventure. Beyond socket options: making the Linux TCP stack truly extensible. In *IFIP International Conference on Networking*, 2019.

[54] Patrick P Tsang, Apu Kapadia, and Sean W Smith. Anonymous IP-address Blocking in Tor with Trusted Computing (Short Paper: Work in Progress). *Proceedings of WATC*, 2006.

[55] Patrick P Tsang, Apu Kapadia, Cory Cornelius, and Sean W Smith. Nymble: Blocking misbehaving users in anonymizing networks. *IEEE Transactions on Dependable and Secure Computing*, 8(2):256–269, 2009.

[56] Philipp Winter. How to report bad relays. https://blog.torproject.org/how-report-bad-relays, 2014.

[57] Philipp Winter. sybilhunter: A Go-based command line tool to discover and analyse Sybil relays in Tor. https://github.com/NullHypothesis/sybilhunter, 2016. Accessed Dec 2020.

[58] Philipp Winter. exitmap: A fast and modular Python-based exit relay scanner. https://github.com/NullHypothesis/exitmap, 2020. Accessed Dec 2020.

[59] Philipp Winter, Richard Köwer, Martin Mulazzani, Markus Huber, Sebastian Schrittwieser, Stefan Lindskog, and Edgar Weippl. Spoiled Onions: Exposing Malicious Tor Exit Relays. In *Privacy Enhancing Technologies Symposium*, 2014.

[60] The Zeek Project. zeek: An open source network security monitoring tool. https://zeek.org/, 2020. Accessed Dec 2020.

A Standard ZKPs used by ZXAD

We now detail the DLE and DLEP zero-knowledge subprotocols used by ZXAD.

A.1 ZKP for Knowledge of Equality of Discrete Logs

Let $\mathbb{G}$ be a cyclic multiplicative group of prime order q and g be one of its generators. Given a tuple of group elements (A, A', B, B'), a prover $\mathcal{P}$ wants to prove the existence of some r such that $A' = A^r$ and $B' = B^r$. Chaum and Pedersen [17] describe a Σ-protocol to prove the knowledge of r, which can be made non-interactive using the the Fiat-Shamir heuristic [24] as follows, denoted $DLE_r[A, A', B, B']$:

i) $\mathcal{P}$ selects $t \xleftarrow{R} \mathbb{Z}_q$ and sets $T_1 = A^t$, $T_2 = B^t$.

ii) $\mathcal{P}$ computes the Fiat-Shamir hash: $c = H(g, T_1, T_2, A, A', B, B') \in \mathbb{Z}_q$.

iii) $\mathcal{P}$ computes $v = t - r \cdot c$ and sends c, v to the verifier $\mathcal{V}$.

iv) $\mathcal{V}$ computes $T_1' = A^v \cdot A'^c$, $T_2' = B^v \cdot B'^c$, and accepts the proof iff $c \stackrel{?}{=} H(g, T_1', T_2', A, A', B, B')$.

We leave off the completeness, soundness, and zero-knowledgeness proof of this Σ-protocol as it is standard.

A.2 ZKP for Knowledge of Equality of Discrete Logs Product

Let $\Lambda = \langle q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_t, e, g_1, g_2, H_1, H_2 \rangle$ be the output of an asymmetric bilinear group generator. Given $A, A' \in \mathbb{G}_1$, $B, B'' \in \mathbb{G}_2$, and $C, D, E \in \mathbb{G}_t$, a prover $\mathcal{P}$ wants to prove the existence of some r_1, r_2 such that $A' = A^{r_1}$, $B'' = B^{r_2}$, and $E = C^{r_1} \cdot D^{r_2}$. We now describe a non-interactive Σ-protocol to prove the knowledge of r_1, r_2 using the the Fiat-Shamir heuristic [24], denoted $DLEP_{r_1, r_2}[A, A', B, B'', C, D, E]$:

i) $\mathcal{P}$ selects $t_1, t_2 \xleftarrow{R} \mathbb{Z}_q$ and sets $T_1 = A^{t_1}$, $T_2 = B^{t_2}$, and $T_3 = C^{t_1} \cdot D^{t_2}$.

ii) $\mathcal{P}$ computes the Fiat-Shamir hash: $c = H(g_1, g_2, T_1, T_2, T_3, A, A', B, B'', C, D, E) \in \mathbb{Z}_q$.

iii) $\mathcal{P}$ computes $v_1 = t_1 - r_1 \cdot c$ and $v_2 = t_2 - r_2 \cdot c$ and sends c, v_1, v_2 to the verifier $\mathcal{V}$.

iv) $\mathcal{V}$ computes $T_1' = A^{v_1} \cdot A'^c$, $T_2' = B^{v_2} \cdot B'^c$, and $T_3' = C^{v_1} \cdot D^{v_2} \cdot E^c$ and accepts the proof iff $c \stackrel{?}{=} H(g_1, g_2, T_1', T_2', T_3', A, A', B, B'', C, D, E)$.

Completeness. $\mathcal{P}$ chooses r_1, r_2, t_1, and t_2 such that it can properly compute v_1 and v_2. Clearly, the Σ-protocol is complete.

Special Soundness. Suppose $\mathcal{P}$ provides two proofs with the same commitment values t_1 and t_2 with challenges c_1 and c_2 respectively. Then we get:

$$v_1 = t_1 - r_1 \cdot c_1 \qquad v_1' = t_1 - r_1 \cdot c_2$$
$$v_2 = t_2 - r_2 \cdot c_1 \qquad v_2' = t_2 - r_2 \cdot c_2$$

We observe that $r_1 = \frac{v_1 - v_1'}{c_2 - c_1}$ and $r_2 = \frac{v_2 - v_2'}{c_2 - c_1}$. Therefore, special soundness is satisfied.

Honest Verifier Zero Knowledge. We define an honest verifier zero-knowledge simulator that is given the challenge c. The simulator chooses $v_1, v_2 \xleftarrow{R} \mathbb{Z}_q$ and sets:

$$T_1' = A^{v_1} \cdot A'^c \qquad T_2' = B^{v_2} \cdot B'^c$$
$$T_3' = C^{v_1} \cdot D^{v_2} \cdot E^c$$

As we can see, the verification equation holds for the simulation and the verifier $\mathcal{V}$ accepts the proof.

B Circuit Token Zero-knowledge Proof

We now prove the soundness and zero-knowledgeness of our circuit token zero-knowledge proof defined in Section 5.1.

B.1 Soundness

First, we prove that our circuit token zero-knowledge proof is sound:

- Statement $\hat{s}_{Tc2}$ (*i.e.*, the zkSNARK proof Π_{T_c}') proves that the client knows some r_2 and a pre-image IP_C under H_2, such that:

$$g_2'' = g_2^{r_2}$$
$$B'' = H_2(IP_C)^{r_2} \tag{1}$$

- Since the client knows r_2 and $\sigma''_{\mathcal{A}}$ is public, the client knows some $\sigma_{\mathcal{A}}$ such that:

$$\sigma''_{\mathcal{A}} = \sigma_{\mathcal{A}}^{r_2} \tag{2}$$

- Now substituting Equation 1 and Equation 2 in Statement $\hat{s}_{T_c 1}$ we get:

$$e(g_1, \sigma_{\mathcal{A}}) = e(A, B) \tag{3}$$

- Therefore, we can conclude that: (i) the prover can unblind $\sigma''_{\mathcal{A}}$ to $\sigma_{\mathcal{A}}$, which is a valid signature on some B by the DirAuth $\mathcal{A}$ (with the secret key corresponding to the public key A) and (ii) B is the hash of some IP_C that the client knows.

B.2 Zero-knowledgeness

Informally, *zero-knowledgeness* guarantees that a valid proof Π_{T_c} does not reveal anything about the witnesses; *i.e.*, r_2 or IP_C.

This is formalized by constructing a simulator that outputs the public values in the same distribution as the honest prover, without knowing the witnesses. We then show that an adversary that distinguishes this simulation from a real proof with non-negligible probability, can be turned into an adversary that breaks an instance of the DDH problem in $\mathbb{G}_2$.

We now prove the zero-knowledgeness of our zero-knowledge proof by defining an honest verifier zero-knowledge simulator. We allow the simulator to have access to a single BLS signature $\langle K, K' = K^{\alpha} \rangle$ from the DirAuths for arbitrary $K \in \mathbb{G}_2$ (not of the simulator's choosing). Note that the simulator does not learn α, nor is the simulated proof claiming to know α.

To output the responses, our simulator first chooses $b, r_2 \xleftarrow{R} \mathbb{Z}_q^*$ and sets $g_2'' = g_2^{r_2}$, $B'' = K^{b \cdot r_2}$, and $\sigma''_{\mathcal{A}} = K'^{b r_2}$. Next, our simulator runs the simulator for the zkSNARK proof on inputs g_2'' and B'' to obtain the public outputs. Finally, our simulator also runs a hash oracle for H_2 which outputs K^b for IP_C and random $r \xleftarrow{R} \mathbb{Z}_q^*$ for all other inputs.

The simulator then sends the token $\sigma''_{\mathcal{A}}$, the simulated zkSNARK proof Π'_{T_c}, and our simulated proof Π_{T_c} to the verifier. As we can see, the verification equation $e(g_1, \sigma''_{\mathcal{A}}) \stackrel{?}{=} e(A, B'')$ holds for the simulation and the exit (*i.e.*, the verifier) accepts the proof. An adversary that distinguishes this simulation from a real proof can be turned into an adversary that given a B can solve an instance of the DDH(g_2, g_2'', B, B'') in $\mathbb{G}_2$.

C Stream Token Σ-Protocol

We now summarize the complete Σ-protocol to prove statements $\hat{s}_{T_s 1}$ and $\hat{s}_{T_s 2}$ (defined in Section 5.2) in Figure 5. We leave off the verification as it is the standard Schnorr-type

StreamTokenProof$(g_1, g_2, Y_1, \ell, n_S, h_1, \ldots, h_{n_S}, \sigma_{\mathcal{A}}, \sigma''_{\mathcal{A}}, g_1', g_2'', Y_1')$

1) C first chooses $\bar{r}_1, \bar{r}_2, \bar{r}_3 \xleftarrow{R} \mathbb{Z}_q$.

2) C then computes the commitments by substituting the random values (chosen in step 1) for the secret terms in statement $\hat{s}_{T_s 2}$:

$$T_2 = g_1^{\bar{r}_1}, \quad T_3 = g_2^{\bar{r}_2}, \quad T_4 = e(Y_1, \sigma''_{\mathcal{A}})^{\bar{r}_1} \cdot e(h_\ell, \sigma_{\mathcal{A}})^{\bar{r}_2}$$

3) Next, to prove statement $\hat{s}_{T_s 1}$, C computes the Borromean [38] OR-proof starting from index ℓ:

- C first sets the commitments for the ℓ^{th} index: $T_{1,\ell,0} = g_1^{\bar{r}_3}, \quad T_{1,\ell,1} = Y_1^{\bar{r}_3}$
- C then computes the Fiat-Shamir hash for index ℓ as follows:

  i) If $\ell \neq 1$, $c_\ell = H(g_1, g_1', Y_1, Y_1', h_\ell, T_{1,\ell,0}, T_{1,\ell,1})$

  else, $c_\ell = H(g_1, g_2, Y_1, h_\ell, T_s, T_{1,\ell,0}, T_{1,\ell,1}, T_2, T_3, T_4, g_1', Y_1', g_2'', \sigma''_{\mathcal{A}})$

- Next C chooses $\ell - 1$ "fake" response values for all indices $i \neq \ell$: $V_{r_{1a},1} \ldots, V_{r_{1a},\ell-1}, V_{r_{1a},\ell+1} \ldots, V_{r_{1a},n_S} \xleftarrow{R} \mathbb{Z}_q$.
- C then computes the the commitments and the Fiat-Shamir hash in a ring ordering starting from index $i = \ell + 1$ to n_S and then from $i = 1$ to $\ell - 1$ as follows:

  i) C first sets the commitments for the i^{th}index:

$$T_{1,i,0} = g_1^{V_{r_{1a},1}} \cdot g_1'^{c_{i-1}} \qquad T_{1,i,1} = Y_1^{V_{r_{1a},1}} \cdot (Y_1' \cdot h_i^{-1})^{c_{i-1}}$$

  ii) If $i \neq 1$, $c_i = H(g_1, g_1', Y_1, Y_1', h_i, T_{1,i,0}, T_{1,i,1})$

  else, $c_1 = H(g_1, g_2, Y_1, h_1, T_s, T_{1,\ell,0}, T_{1,\ell,1}, T_2, T_3, T_4, g_1', Y_1', g_2'', \sigma''_{\mathcal{A}})$

- Finally, C sets the response for the ℓ^{th} index: $V_{r_{1a},\ell} = \bar{r}_3 - r_1 \cdot c_{\ell-1}$

4) Finally, C sets the response values for all other secret terms using the Fiat-Shamir hash c_1 produced above:

$$V_{r_{1b}} = \bar{r}_1 - r_1 \cdot c_1 \qquad V_{r_2} = \bar{r}_2 - r_2 \cdot c_1$$

5) C sends the token T_s and the proof $\Pi_{T_s} = \langle g_1', Y_1', V_{r_{1a},1} \ldots, V_{r_{1a},n_S}, V_{r_{1b}}, V_{r_2} \rangle$ to the Tor exit.

Figure 5: Σ-protocol to prove that the stream token T_s is well formed

proof verification (defined in Section 2.2) and is straightforward.

C.1 Soundness

Informally, *soundness* guarantees that only clients with a well-formed token T_s can generate a valid proof Π_{T_s}, that will be accepted by the verifier (*i.e.*, the Tor exit). We now prove that our Σ-protocol is sound:

- Statement s_{T_s1} proves that the client knows some r_1 and ℓ, $1 \leq \ell \leq n_S$ such that $g_1^{'} = g_1^{r_1}$ and $Y_1^{'} = Y_1^{r_1} \cdot h_\ell$.
- Statement s_{T_s2} proves that the client knows some r_2 such that $g_2^{''} = g_2^{r_2}$, and

$$\begin{aligned} & e(Y_1^{'}, \sigma_{\mathcal{A}}^{''}) \;=\; e(Y_1, \sigma_{\mathcal{A}}^{''})^{r_1} \cdot T_s^{r_2} \\ & \Leftrightarrow\; e(Y_1^{r_1} \cdot h_\ell, \sigma_{\mathcal{A}}^{''}) \;=\; e(Y_1^{r_1}, \sigma_{\mathcal{A}}^{''}) \cdot T_s^{r_2} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{(since } Y_1^{'} = Y_1^{r_1} \cdot h_\ell\text{)} \\ & \Leftrightarrow\; e(h_\ell, \sigma_{\mathcal{A}}^{''}) \;=\; T_s^{r_2} \qquad (4) \end{aligned}$$

- Since $\sigma_{\mathcal{A}}^{''}$ is public, the client knows some $\sigma_{\mathcal{A}}$ such that $\sigma_{\mathcal{A}}^{''} = \sigma_{\mathcal{A}}^{r_2}$.
- Now substituting $\sigma_{\mathcal{A}}^{''} = \sigma_{\mathcal{A}}^{r_2}$ in Equation 4 we get:

$$T_s \;=\; e(h_\ell, \sigma_{\mathcal{A}}) \qquad (5)$$

- Finally, from Equation 1 and Equation 3 we get:

$$\sigma_{\mathcal{A}} \;=\; B^{\alpha} \qquad (6)$$

- Therefore, from Equations 4–6 we can conclude that T_s is well formed.

C.2 Zero-knowledgeness

Informally, *Zero-knowledgeness* guarantees that a valid proof Π_{T_s} does not reveal anything about the witnesses; ℓ, r_1, or r_2.

This is formalized by constructing a simulator that outputs the public values in the same distribution as the honest prover, without knowing the witnesses. We then show that an adversary that distinguishes this simulation from a real proof with non-negligible probability, can be turned into an adversary that breaks an instance of the DDH problem in $\mathbb{G}_1$.

We now prove the zero-knowledgeness of our Σ-protocol. We first define an honest verifier zero-knowledge simulator that is given the challenge c_1 for the Schnorr-type proof (defined in Section 2.2).

To output the responses, our simulator first chooses random $V_{r_{1b}}, V_{r_2}, V_{r_{1a},1} \ldots, V_{r_{1a},n_S}, \bar{r}_1, \bar{r}_2 \xleftarrow{R} \mathbb{Z}_q$ and sets $g_1^{'} = g_1^{\bar{r}_1}$, $g_2^{''} = g_2^{\bar{r}_2}$. Next, it chooses $L \xleftarrow{R} \mathbb{G}_1$ and $\sigma_{\mathcal{A}} \xleftarrow{R} \mathbb{G}_2$ and sets $T_s = e(L, \sigma_{\mathcal{A}})$, $Y_1^{'} = Y_1^{\bar{r}_1} \cdot L$, and $\sigma_{\mathcal{A}}^{''} = \sigma_{\mathcal{A}}^{\bar{r}_2}$. Finally, it finishes the simulation of the proof as follows:

- For the statement s_{T_s1}, our simulator runs the simulator for the Borromean ring OR-proof on $\bigvee_{i=1}^{n_S} DLE\left[g_1, g_1^{'}, Y_1, Y_1^{'} \cdot h_i^{-1}\right]$ with the given challenge c_1 and obtains the responses $V_{r_{1a},1} \ldots, V_{r_{1a},n_S}$. It uses the Σ-protocols for the Borromean ring OR-proof [38] and the Chaum and Pedersen [17] proof for knowledge of equality of discrete logs. We omit the zero-knowledgeness proofs for these Σ-protocols as they are standard.
- For statement s_{T_s1}, our simulator runs the simulator for $DLEP\left[g_1, g_1^{'}, g_2, g_2^{''}, e(Y_1, \sigma_{\mathcal{A}}^{''}), T_s, e(Y_1^{'}, \sigma_{\mathcal{A}}^{''})\right]$ with the challenge c_1 and obtains the responses $V_{r_{1b}}, V_{r_2}$. The zero-knowledgeness of *DLEP* is proved in Appendix A.2.

The simulator then sends the token T_s and the simulated proof Π_{T_s} to the verifier. An adversary that distinguishes this simulation from a real proof can be turned into an adversary that given a h_ℓ, $1 \leq \ell \leq n_S$ can solve an instance of the DDH$(g_1, g_1^{'}, Y_1, Y_1^{'} \cdot h_\ell^{-1})$ in $\mathbb{G}_1$.