GoAT: File Geolocation via Anchor Timestamping

Deepak Maram[1,2⋆], Mahimna Kelkar[1], Iddo Bentov[1], and Ari Juels[1]

[1] Cornell Tech
[2] Mysten Labs

Abstract. Decentralized storage systems are a crucial component of the rapidly growing blockchain ecosystem. They aim to achieve robustness by proving that they store multiple replicas of every file. They have a serious limitation, though: They *cannot* prove that file replicas are spread across distinct systems, e.g., different hard drives. Consequently, files are vulnerable to loss in a single, locally catastrophic event.

We introduce a new primitive, *Proof of Geo-Retrievability* or PoGeoRet, that proves that a file is located within a strict geographic boundary. Using PoGeoRet, one can, for example, prove that a file is spread across several distinct geographic regions—and by extension across multiple systems, e.g., hard drives. We define what it means for a PoGeoRet scheme to be complete and sound, extending prior formalism in key ways.

We also propose GoAT, a practical PoGeoRet scheme to prove file geolocation. Unlike previous geolocation systems that only offer nominal geolocation guarantees and require dedicated anchors, GoAT geolocates provers using any timestamping server on the internet with a fixed, known location as a geolocation anchor. GoAT's geolocation guarantees directly depend on the physical constraints of the internet, making them very reliable. GoAT internally uses a communication-efficient Proof-of-Retrievability (PoRet) scheme in a novel way to achieve constant-size PoRet-component in its proofs.

We validate GoAT's practicality by conducting an initial measurement study to find usable anchors and perform a real-world experiment. The results show that a significant fraction of the internet can be used as anchors and that GoAT achieves geolocation radii as low as 500km.

1 Introduction

Decentralized systems are a rapidly expanding form of computing infrastructure. Blockchain systems in particular have enjoyed considerable recent popularity and constitute a $1.5 trillion market at the time of writing [2]. Many decentralized applications, ranging from non-fungible tokens (NFT) [19] to retention of blockchain state [3], require a reliable bulk storage medium. As blockchains have limited innate storage capacity, there is thus a growing demand for purpose-built decentralized storage systems, of which a number have arisen, such as IPFS [13], Filecoin [32], Sia [41], Storj [33], etc.

⋆ Corresponding author: sm2686@cornell.edu

Like today's cloud storage services (e.g., Amazon S3 [9]), decentralized storage systems typically achieve robustness by replicating files. With this approach, even if some replicas become unavailable, others can be used to fetch files. To help ensure trustworthy storage of replicas, decentralized file systems require storage providers to prove retention of file replicas. As a notable example, Filecoin [14] uses a protocol called Proof of Replication (PoRep) [22] for this purpose. Related systems such as Sia, Storj, etc., use alternative techniques.

While a PoRep or related proof system can prove the existence of multiple copies of a file, however, its robustness assurances are limited. This is because such systems *cannot ensure that file replicas reside on independent devices or systems.* If all file replicas are stored on the same hard disk, for example, damage to that one device can destroy the file.

In this paper, we explore an alternative approach to building robust decentralized storage systems: proving that file replicas *reside in distinct geographical regions.* For example, one may wish to prove that three replicas of a file are present in the United States, Europe, and Asia respectively. Such a proof automatically implies the property ensured by existing techniques such as PoReps, namely the existence of three distinct replicas of the file. It also ensures much stronger properties than a proof of replication alone, namely that file replicas are *stored on distinct devices* and *in distinct physical locations.* These additional properties imply that the file can survive device failures, destructive local events (e.g., natural catastrophes), etc. Thus the ability to prove replica geolocation can greatly improve robustness in decentralized storage systems. Geolocation-based proofs can also incur *substantially lower resource costs* than techniques like PoReps, as we show in this paper.

Beyond rendering decentralized storage more robust, proving storage location is useful in other settings. For example it can help prove compliance with laws specifying localized storage of certain forms of data, e.g., [26].[3] It can also be used by CDN providers to prove that they are serving data from geographically distributed locations according to a claimed policy.

In brief, the goal of our work here is to build protocols to prove that a given file replica is stored within a strictly-bounded geographical region. Our main building block for these protocols is a primitive we call a *Proof of Geo-Retrievability* (PoGeoRet). A PoGeoRet involves a single prover proving to a number of verifiers that it holds a file replica in a given geographical region. To ensure the practicality of our PoGeoRet designs we consider here, we focus on proofs involving relatively large geographical regions (e.g., thousand-km radius), which suffices for applications such as file replication.

We introduce a formal definition of PoGeoRets in this paper, and propose, implement, and experimentally validate a PoGeoRet system called GoAT. GoAT creates publicly verifiable file-replica geolocation proofs. GoAT proofs can thus be consumed by a multiplicity of verifiers and can be used to construct a system

[3] Some nations only require that a copy of data be stored locally whereas more stricter laws make transferring data abroad illegal [26]. Our techniques suffice for the former but the latter would additionally require the use of trusted hardware.

that ensures the presence of file replicas in desired locations even in the presence of some dishonest verifiers.

The Anchor Model: To avoid the undesirable assumptions of previous geolocation systems, we explore a model for GoAT that relies on a collection of servers called *anchors*. An anchor is a server with a *publicly announced location* that emits digitally signed *timestamps* on queries. That is, an anchor has an API that returns the current time along with a signature over the time and any value sent by a client. Crucially, the main job of an anchor is *not* to geolocate entities directly, but only to provide timestamps. Anchors can therefore be *distinct from verifiers*. Anchors can be purpose-built for a GoAT instance. We also show that it is possible to use *existing, unmodified* servers, e.g., TLS 1.2 or Roughtime [7] servers, as anchors.[4] Thus it is possible with some tradeoffs to realize GoAT with *today's internet infrastructure*.

1.1 Conceptual Starting Point: Proving Geolocation

In GoAT, a prover must prove proximity of a file it stores to an anchor. We first explain how a prover can prove its own geolocation. Then we explain how these techniques are embellished in GoAT to prove geolocation of a file.

To prove geolocation, GoAT uses a simple, well known technique: The prover pings the anchor successively to get two timestamps t_1, t_2 (see Fig. 2). If the prover is indeed situated in proximity to the anchor, then the timestamps will not differ by much, i.e., $t_2 - t_1 < \varDelta$, for a small predetermined threshold $\varDelta$. Note that if the pings are truly *sequential*, i.e., the second ping happens after the first is completed, a prover can spoof a *higher* distance from an anchor just by delaying t_2, but cannot spoof a *lower* distance, and thus cannot falsify proofs of proximity.

Realizing proofs of geolocation: Even before incorporating files into geolocation proofs, several challenges arise.

First, ensuring correct geolocation proofs involves some subtleties. For example, as we have noted, it is critical that the prover's pings of an anchor be *sequential*. Otherwise a dishonest prover can interleave pings and shrink $t_2 - t_1$ arbitrarily. Thus GoAT proofs are designed in such a way that later pings by the prover depend cryptographically on earlier ones.

An especially important challenge is to be compatible with existing infrastructure, i.e., using existing servers as anchors. TLS 1.2 servers, for instance, only provide second-level timestamps, which is insufficient as network round-trip times are on the order of milliseconds. We address this challenge by introducing an *amplification* technique: Instead of pinging twice, a prover pings the anchor repeatedly with related challenges over an extended time interval, e.g., 1 second.

Another challenge is identifying usable anchors. Many TLS 1.2 servers, for instance, do not return accurate time or have a unique location, needed for a prover to prove geolocation. We conduct an initial measurement study of the Alexa top 1M list to identify a broad network of usable anchors.

[4] TLS 1.2 is still almost universally supported. Roughtime is an emerging protocol adopted by Google, Cloudflare, etc.

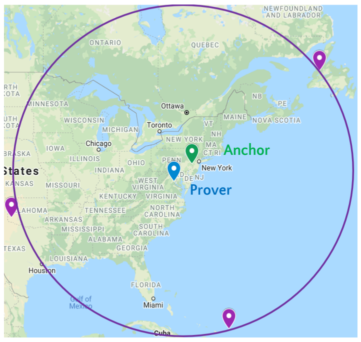

Fig. 1: Prover and (Roughtime) anchor are situated 300km apart. GoAT's region of uncertainty is a circle of radius 2000km (purple) proving that the prover's file replica is in eastern North America.

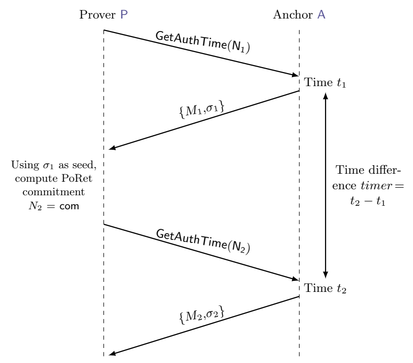

Fig. 2: The geo-commitment protocol.

A key requirement is reliable geolocation guarantees. GoAT bases its guarantees on physical network limitations making no assumptions about the underlying network topology. Concretely, GoAT's geolocation accuracy depends on two factors: (i) the real spherical distance between the anchor and the prover, and (ii) a parameter ω that denotes how fast an adversary can send and receive messages compared to the internet speeds of an honest party. We derive ω by conservatively assuming that the adversary can send messages at light speed and fixing a realistic lower speed (set based on empirical data) that any honest prover can meet.

The use of existing servers as anchors also requires us to consider an *economically rational adversary* that will not incur bandwidth costs in excess of the revenue it receives for storage (Sec. 3.3). In brief, this is necessary because of the passive nature of anchors in GoAT, which, in contrast to prior research, lack the capability to challenge the prover at random times. Nevertheless, we assert that this approach is a more pragmatic solution to geolocation than mandating the establishment of a new geographically distributed network of anchors.

1.2 Geolocating Files: GoAT

Recall that our goal in GoAT is not to geolocate servers, but *files*. Building on basic geolocation proofs as described above, a straightforward idea is to interleave the file into the prover's anchor pings. But this approach only works for extremely small files due to limits on input sizes to anchors, typically just hundreds of bytes. Given that GoAT aims to geolocate files of significantly larger sizes, this approach is not practical.

So the key idea in GoAT is to interleave what is called a *Proof of Retrievability* (PoRet) [29,38]. A PoRet is compact, efficient proof that a full file replica is indeed stored.

Making GoAT work: A key challenge in interleaving geolocation proofs with PoRets arises from the computational latency introduced by PoRets. It is essential to minimize the latency in GoAT between the two anchor pings, as this latency impacts geolocation accuracy. Each *millisecond* of computational latency caused by PoRet computation degrades geolocation accuracy by *about 100km*. Our solution to this problem is to introduce techniques for computing a (computationally lightweight) commitment to the randomness in a PoRet proof between two pings, *before computing the PoRet proof itself*. Our technique improves geolocation accuracy by an order of magnitude in some cases.

Yet another challenge is reducing the size of GoAT proofs. Due to a combination of amplification and proof accumulation across epochs and anchors, the proof sizes quickly blow up, even with use of a communication-efficient PoRet (e.g., [38]). Using techniques that combine vector commitments and compression across proof instances, we manage to achieve *at least 3x reduction in proof sizes* (potentially unbounded). We do in a GoAT variant called GoAT-P by introducing a new homomorphic commitment scheme that relies on a new knowledge assumption closely related to the *Knowledge-of-Exponent Assumption* (KEA), specifically KEA1 [12]. (Another variant of GoAT called GoAT-H sacrifices compactness in favor of a standard commitment scheme using hash functions.)

Given use of these techniques, GoAT can geolocate files regionally using already existing infrastructure. For instance, Figure 1 illustrates GoAT's file geolocation capabilities using a local Roughtime server.

1.3 Contributions and Paper Organization

We introduce preliminaries in Sec. 2. Our contributions are:

1. GoAT: We introduce our Proof of Geo-Retrievability (PoGeoRet) protocol GoAT (Sec. 4). GoAT leverages the Shacham-Waters PoRet and timestamping anchors. Using several new techniques to minimize computational latency and proof sizes, we make GoAT practical.
2. *Implementation and Evaluation*: To demonstrate practicality, we prototype GoAT (GoAT-P) and run a small real-world experiment using 10 TLS / Roughtime anchors (5 each in the US and UK) for over a week. GoAT's prove and verify protocols execute in just a few seconds, with proof sizes of a few hundred KB. GoAT can work with geolocation radii lower than 1000km, smaller than required for applications like file replication (Sec. 5).

   We have released GoAT as an open-source tool.

Organization: We cover the necessary preliminaries in Sec. 2. Section 3 introduces Proofs of Geographic Retrievability and the system model. Section 4 specifies the GoAT protocols. Sec. 5 presents the implementation and evaluation details, and we conclude with Sec. 6.

Due to lack of space, the formal security definitions are in App. A and GoAT's security is defined in App. C. Finally, we prove security in App. D.

1.4 Related Work

Several works aim to prove correct file storage by a storage provider e.g., PoRet [38,29], Proof of Data Possession [11,25] and more recently Proof of Replication [22,18,23,15].

To the best of our knowledge, only few works [16,43] aim to prove file location. [16] works with small files as they rely on directly fetching file parts and leave open the task of combining a PoRet with a proof-of-location (PoL). [43] combines Shacham-Waters PoRet scheme with a PoL, making it the closest to our work. But they make benign assumptions about storage providers, e.g., providers operate at normal network speeds. In contrast, GoAT considers faster speed-of-light providers and yet achieves geolocation accuracy similar to [43]. Digging deeper, this is due to our novel use of fast PoRet commitments whereas [43] naïvely combines the SW PoRet and PoL protocols. [43] also entrusts anchors with proof verification making the use of legacy anchors impossible unlike GoAT.

Many works consider geolocating servers [30,42,31,28]. Of these, only [31] allows verifying node location in decentralized settings, but [31] operates in a weaker model where an adversary controls a randomly sampled set of nodes.

Finally, many prior works [30,42,16] have used distance-bounding like techniques for geolocation. Our protocol is also a novel application of the same underlying technique: typically distance bounding works by verifier sending a challenge and measuring the time taken for a prover to respond, whereas we leverage public timestamping servers to track time instead of a verifier.

2 Preliminaries

Authenticated time protocols: We are interested in *authenticated* time protocols, i.e., the timestamp must be digitally signed. Two main options exist today:

(1) TLS 1.2. Some TLS 1.2 servers [20] embed the current time in seconds into the first 8 bytes of the "server random" value. This value is then signed and sent to the client as part of TLS 1.2 key exchange. The receiving party verifies the signature using the server's certificate.

This functionality has always been an informal practice, and is not specified in the TLS 1.2 RFC, but is widespread. We found about 1/5 of Alexa top-500 hosts supported this technique. Finally, this method does not work with TLS 1.3, as the specification specifically deprecates it. In practice though, TLS 1.3 adoption is only growing slowly. And TLS 1.2 is expected to be supported by most websites for the foreseeable future; e.g., in July 2022, 99.8% of around 1M sites surveyed were found to support TLS 1.2 [36].

(2) Roughtime. Roughtime [7] is a recently developed authenticated time protocol. At the time of writing, we are aware of four providers hosting Roughtime—Cloudflare [35], Google, Chainpoint and int08h. Roughtime servers provide a highly precise timestamp in microseconds (μs) signed with a fast signature scheme (EdDSA). As the name "Roughtime" suggests, the protocol is only designed to provide a roughly accurate time, say within 10 seconds of the true time, unlike say NTP. Note that GoAT does not need accurate absolute time.

Proof of Retrievability: Proof of Retrievability [29] schemes enable a prover to prove knowledge of a complete file replica in a communication-efficient manner. GoAT requires a publicly verifiable PoRet scheme—we use the communication-efficient Shacham-Waters (SW) [38]. Figure 9 in the Appendix shows the API for a PoRet scheme. A special feature of GoAT is the introduction of an additional functionality in a PoRet. This functionality, called PoRet.Commit, commits to randomness for use in a (future) PoRet proof. We introduce PoRet.Commit to enable fast prover interaction with a timestamping service, and thus require that it be: (1) quickly computable (within a few milliseconds), and (2) compact. We specify our construction of PoRet.Commit later.

3 Model

We introduce Proofs of Geographic Retrievability along with informal definitions for completeness and soundness in Sec. 3.1. We specify our model in Sec. 3.2 and conclude with some practical modeling choices in Sec. 3.3.

3.1 Formalizing Proofs of Geographic Retrievability

A Proof-of-Geographic-Retrievability (PoGeoRet) scheme includes three parties: a *user* (U) who owns a file F, a *storage provider* or prover (P) that commits to storing F for a specified duration at a specified location, and an *auditor* or verifier (V) that verifies P's claims. (Of course, a decentralized system will include many instances of each party type.)

We define a PoGeoRet for a general setting in which a target file F is stored as a publicly accessible plaintext.

A user U that wants a file F to be stored near a particular location runs the setup protocol (PoGeoRet.Setup) on F to generate an encoded file F^*. U then gives F^* to a storage provider P situated near the desired location. The public parameters *pp* are published, e.g., on a blockchain. The file handle η is unique for each file, for example, it could be a hash of F.

A PoGeoRet protocol runs in *epochs*. During each epoch, the provider P computes a *Proof of Geo-Retrievability* using the Prove protocol. All PoGeoRet schemes must use time to distinguish between a challenge answered with a local file versus another answered with a file fetched from afar. That is, some or all of the Prove protocol must be timed by the verifier. Accordingly, Verify takes the time *timer* taken to respond and a proof of geo-retrievability π^{geo} as inputs and outputs a single bit indicating "success" or "failure".

The key ingredient in Prove enabling a file geolocation proof is the sub-function ComFrag. It takes a file fragment $\boldsymbol{\mu}$ as input and outputs a commitment of it. Intuitively, a file fragment represents the bare minimum data related to F that a prover must have used when running Prove. Looking ahead, in GoAT, a file fragment is a vector of data blocks derived from F and ComFrag is a vector commitment scheme.

A PoGeoRet must also specify an extraction algorithm $\mathsf{Extract}$ that can recompute the file F from the prover's responses. $\mathsf{Extract}$ will be used to model extraction in the soundness definition of a PoGeoRet in a way largely similar to prior works [29,38]. A key difference is that $\mathsf{Extract}$ needs to follow a specific design; it must be composed of two algorithms: $\mathsf{Extr.Derive}$, which, interacts with the prover and outputs a list of file fragments μ^{all} and $\mathsf{Extr.Assemble}$, which recomputes the file F from the fragments. We assume that during $\mathsf{Extr.Derive}$, the prover can be *rewound*, as in, e.g., [29,38].

The PoGeoRet API is shown in Fig. 5 in the Appendix. Note that above definitions require interaction between prover and verifier, whereas our main goal is for GoAT to operate non-interactively. The non-interactive PoGeoRet API only requires minor modifications (Fig. 6).

Modeling geolocation: We model geolocation in a PoGeoRet using a metric space [5] ($\mathcal{M}$,dist) where $\mathcal{M}$ is the full set of possible storage locations and dist is a distance metric on $\mathcal{M}$. As an example, $\mathcal{M}$ could be the set of all points on a sphere (e.g., the Earth) and dist the spherical distance function.

For $L \in \mathcal{M}$, we define a *region* $R = (L;\delta)$ as the set of all $L' \in \mathcal{M}$ that satisfy $\mathsf{dist}(L,L') \leq \delta$. For example, when $\mathcal{M}$ models points on a sphere, regions represent circles on the surface. For simplicity, we will only consider such circular regions.

We want a PoGeoRet scheme to facilitate storage of files in a target region $R^{\mathsf{target}} = (L;\delta)$ where δ is a radius that captures the breadth of the target region. Our definition allows for any arbitrary δ.

Region of uncertainty: We define a *Region Of Uncertainty* (ROU), denoted R^{rou}, to capture the permitted noise in an attained geolocation guarantee. A PoGeoRet scheme for a given R^{target} ensures that files are stored inside the larger region R^{rou}. By relaxing geolocation beyond a small target region, an ROU helps eliminate spurious proof failures.

Suppose we wish to support file storage in New York City. Then the target region is $R^{\mathsf{target}} = (L;\delta)$ where, e.g., $L = (40.73°,-73.93°)$ and $\delta = 30$km. Suppose that we are willing to tolerate noise in proofs up to the point where we ensure that files are at most 500km from New York. The desired region of uncertainty then is $R^{\mathsf{rou}} = (L;500\text{km})$.

Desired properties: Like any security protocol, a PoGeoRet must satisfy two basic properties: *completeness* and *soundness*. Completeness means that the PoGeoRet scheme must succeed for any honest prover storing the file in R^{target}. Soundness means that any dishonest prover either not storing the complete file or storing it outside a permitted geographic boundary (R^{rou}) should be detected with high probability.

The formalization for PoGeoRet soundness is similar in spirit to that for PoRet but leads to interesting new subtleties. Intuitively, a PoGeoRet is sound if acceptance by a verifier means that a file F can be extracted from the prover, similar to prior PoRet formalism [38]. The key difference for a PoGeoRet is that successful extraction must now be possible *from the target location.* To capture the notion of file location, we introduce a *location-specific commitment oracle.* This oracle models the PoRet commitment function and tracks queries made to it from within the target region of uncertainty (R^{rou}). A PoGeoRet is sound if the file fragments seen by the commitment oracle are enough to recompute the file.

Due to lack of space, we relegate the formal security definitions of completeness and soundness to App. A.

3.2 GoAT System Model

Network model: We approximate the Earth to be a sphere. The metric space ($\mathcal{M}$, dist) is defined by the set of all locations on earth ($\mathcal{M}$) and the spherical distance function (dist). We assume that the maximum network speed attainable by an adversary is S_{max}. The minimum speed required for storage providers is S_{min}. The ratio $\omega = S_{\mathsf{max}}/S_{\mathsf{min}}$ is the adversary's *network advantage*. These parameters need to be set based on empirical measurements (see Sec. 5). Honest providers only need to attain S_{min} *transiently*. For example, if the interval length is $\beta = 1$hr, good connectivity for a few seconds every hour suffices.

Our network model includes a small per-transmission connection setup cost t_{setup}; empirically, such costs dominate round trip times for nearby locations. The expected maximum time for a round trip between locations L_1 and L_2 is given by $\mathsf{rtt}_{\mathsf{max}}(L_1,L_2) = (2\mathsf{dist}(L_1,L_2)/S_{\mathsf{min}}) + t_{\mathsf{setup}}$.

Anchors: GoAT leverages existing internet servers called *anchors*. Anchors must provide an authenticated time API and have a static known location. Their clocks must provide relative time with negligible clock drift.

Anchors serve time through the GetAuthTime API. It takes as input a nonce N and returns a transcript $T = \{M,\sigma\}$. $M = \{t,N\}$ is a message containing the time t and nonce N (and potentially other data, as discussed later); σ is a signature $\sigma = \mathsf{Sig}_{sk_{\mathsf{A}}}(M)$ on M, for anchor key pair (sk_{A}, pk_{A}). We assume an authoritative list of well placed anchors $\mathcal{T}$ selected for trustworthiness and reliability.

The timestamp resolution Γ_{A} of an anchor A is defined as the smallest (non-zero) difference between any two timestamps. GoAT supports anchors of all resolutions, although smaller resolution leads to better performance.

Adversarial model: The adversary $\mathcal{A}$ controls the storage provider P and a fraction of anchors such that there exist a local honest majority (see Sec. 4.2 for an explanation). We assume that the user U is honest.

Storage: We assume storage providers use SSDs (we do not support HDDs; see [34]) since inbuilt parallelism allows SSD seek times to be only a few ms [6].

Operational model: Time is organized into *epochs* each containing I *intervals* of length β. A prover P must generate at least one proof per interval. Verifier V checks P's proofs one per epoch, i.e., once per interval of time of length $I \times \beta$.

3.3 Flexible-challenge model: Eliminating random audits

Previous works on geolocation [31] usually consider a *random-challenge model*, in which a verifier pings a prover at random times. A limitation of that model is that it requires a network of verifiers that interact with and challenge provers. However, this network of verifiers would need to be *geo-distributed*, i.e., have a presence close to any potentially asserted geolocation of a prover. Otherwise, network delays could potentially give a prover an opportunity to cheat and download file contents upon being challenged.

Thus, while such auditing is practical for GoAT in principle, it carries significant overhead. We instead favor a more practical deployment option using what we call the *flexible-challenge model*. Here, instead of being randomly audited, a prover generates its own proofs. The prover can do so at any time it chooses but at least once every pre-specified *auditing interval*.

A prover P can, of course, try to cheat by taking advantage of this flexibility to download F to a device in R^{rou} before generating a proof. We show, however, that a *rational* prover, i.e., one who is financially motivated, will not cheat this way—provided that auditing intervals are sufficiently short. In a nutshell, to avoid storing F in R^{rou}, P would need to download F frequently. It would thereby incur bandwidth costs that exceed its revenue from file storage.

Define a parameter ϕ called the *bandwidth overhead*. This is the ratio between the cost of (per-byte) storage over an interval and the cost of (per-byte) bandwidth. (For simplicity, we assume linear costs.) So a rational adversary will not download more than $\phi|F^*|$ per interval on average, as the result would be a loss of revenue. We find that for short intervals, typically $\phi \ll 1$ (see App. A.3).

4 The GoAT protocol

We also consider a different PoGeoRet design that uses Merkle Tree PoRet instead of SW. This variant has the benefit of a simpler design and avoids the hardness assumptions required for SW. It has large proof sizes, however, so we defer it to App. H.

We begin with the Shacham-Waters scheme which is heavily used in GoAT. SW is a PoRet, i.e., scheme in which a prover P stores a file F, transformed into F^* by a file owner U. P proves to a verifier V (not necessarily the file owner) that it is possible to retrieve F from the prover in full. SW's API are explained now (see Fig. 8 in the Appendix for a concise description).

Let $\mathbb{Z}_p$ be the support of a group $\mathbb{G}$. Key generation in SW involves the file owner generating a secret key $sk \in \mathbb{Z}_p$, with corresponding public key $pk \in \mathbb{G}$.

Let F be an erasure-coded file. SW.Setup involves dividing F into n blocks, with each block further divided into s sectors. The set of all sectors is $\{m_{ij}\}_{1 \le i \le n, 1 \le j \le s}$. Each sector m_{ij} is a symbol in $\mathbb{Z}_p$. To prepare F for storage, the file owner executes function SW.Setup on F, yielding a transformed file F^* for the prover. To do so, the file owner picks s group elements $\{u_j\}_{j=1}^{s}$ at random as public parameters pp and computes aggregate signatures $\{\sigma_i\}_{i=1}^{n}$ over all the sectors in each block, $\forall i \in [1,...,n]$, $\sigma_i \leftarrow (\mathsf{H}(i) \prod_{j=1}^{s} u_j^{m_{ij}})^{sk}$, where H is a hash function. The file owner outputs $F^* = (F, \{\sigma_i\}_{i=1}^{n})$ and file handle $\eta = \mathsf{H}(F^*)$ for the prover to store.

A verifier challenges P by invoking SW.Chal, which outputs a set of l challenges $\mathcal{S}$. Each challenge consists of a randomly chosen block number $c_k \leftarrow\$ [1, \dots, n]$ and a coefficient $v_k \leftarrow\$ \mathbb{Z}_p$, i.e. $\mathcal{S} = \{c_k, v_k\}_{k=1}^{l}$. The prover invokes SW.Prove to respond, computing several linear combinations $\forall j \in [1,...,s]$, $\mu_j \leftarrow$

$\Sigma_{k=1}^{l} v_k m_{(c_k)j}$ and an aggregate signature $\sigma \leftarrow \prod_{k=1}^{l} \sigma_{(c_k)}^{v_k}$. In summary, $(\boldsymbol{\mu},\sigma) \leftarrow$ SW.Prove$(\eta,\mathcal{S})$ for $\boldsymbol{\mu} = \{\mu_j\}_{j=1}^{s}$. (Note that bold face denotes vectors.)

SW.Verify verifies the proof $(\boldsymbol{\mu},\sigma)$ using pk and other public parameters.

Modifications to SW in GoAT: GoAT adds a function SW.Commit that takes the same inputs as SW.Prove but only outputs a *vector commitment* (VC) of $\boldsymbol{\mu}$.

We consider two variants of SW, SW-P and SW-H, that differ in the choice of VC. The main one, SW-P, uses the *correlated Pedersen scheme*, where two normal Pedersen commitments are computed, but with correlated bases. That is, given a scalar b, two sets of correlated bases $\mathbf{h}_1 \in \mathbb{G}^s$, $\mathbf{h}_2 = \mathbf{h}_1^b$, and a vector $\boldsymbol{\mu} \in \mathbb{Z}_p^s$, the VC of $\boldsymbol{\mu}$ is $(\mathbf{h}_1^{\boldsymbol{\mu}}, \mathbf{h}_2^{\boldsymbol{\mu}}) \leftarrow \mathsf{CPVC.Commit}(\boldsymbol{\mu})$. (The exponentiation operation between the vectors denotes multi-scalar multiplication.) Framing it as part of SW-P, we have $(\mathbf{h}_1^{\boldsymbol{\mu}}, \mathbf{h}_2^{\boldsymbol{\mu}}) \leftarrow$ SW-P.Commit$(\eta,\mathcal{S})$. The other variant SW-H uses a hash function, so $\mathsf{H}(\boldsymbol{\mu}) \leftarrow$ SW-H.Commit$(\eta,\mathcal{S})$. The verification functions (SW-P.Verify and SW-H.Verify) take a commitment of $\boldsymbol{\mu}$ as an extra input and verify that $\boldsymbol{\mu}$ provided in the proof matches its commitment.

4.1 GoAT protocols

We now present GoAT protocol for high-resolution anchors (i.e., $\Gamma_{\mathsf{A}} \leq 1$ms). A real-world example is an anchor using Roughtime [7], which has a 1μs resolution. Low-resolution anchors are considered in Sec. 4.2. Two variants of GoAT arise, depending on the PoRet scheme used. The first, GoAT-H, uses the SW-H PoRet scheme, and admits a security proof in the random oracle model. The second, GoAT-P, uses the SW-P PoRet scheme. It has more compact proofs than GoAT-H, but has security based on a new assumption (see Definition 3 in App. C; roughly, this KEVA assumption extends the commonly used KEA1 [12] assumption for a vector of elements). The two schemes are largely similar, so we only present GoAT-P and defer GoAT-H to App. B.

Setup (Setup): User U runs PoRet setup (SW.Setup) over file F to generate transformed file F^*, file handle η, and the public parameters pp. Then U picks a storage prover P located at an admissible location L_{P} and sends $\{F^*, \eta, pp\}$ to P.

As noted before, we assume that a set of anchors $\mathcal{T}$ is predetermined; let $\mathsf{A} \in \mathcal{T}$ be one such anchor, located at L_{A}. For simplicity, in this section, we assume anchors are trusted and thus that it suffices to use the single anchor A.

Proof generation (Prove): P generates a proof of geo-retrievability in two phases. In the first, *geo-commitment generation phase*, P interacts with A to obtain PoRet challenges and uses them to generate a PoRet commitment. This phase is run once per interval. In the second, *PoRet computation phase*, run once per epoch, P computes the full PoRets. We now give details on the two phases.

Geo-commitment generation phase (GeoCommit): The key idea in GoAT is to sandwich the file access operation between successive pings to the anchor. The prover uses the signature returned in the first ping as a PoRet challenge and computes a PoRet commitment. The commitment is set as the nonce in the second ping. Fig. 2 depicts the geo-commit protocol laid out as follows. Note that all the steps below are run by the prover P.

1. *Ping 1:* Send $\mathsf{GetAuthTime}(N_1)$ for random nonce N_1 to A. Receive $T_1 = \{M_1,\sigma_1\}$.
2. *PoRet commitment:* Use σ_1 as randomness to derive a set of challenges $\mathcal{S} \leftarrow \mathsf{SW.Chal}(\eta,pp,\sigma_1)$ and generate a commitment $\mathsf{com} \leftarrow \mathsf{SW\text{-}P.Commit}(\eta,\mathcal{S})$.
3. *Ping 2:* Set $N_2 = \mathsf{com}$ and ping A again via $\mathsf{GetAuthTime}(N_2)$. Receive $T_2 = \{M_2,\sigma_2\}$.

We refer to the pair $C^{\mathsf{geo}} = \{T_1,T_2\}$ as a *geo-commitment*. Note that the PoRet commitment com is embedded in T_2, so we do not explicitly mention it. By the end of an epoch (or I intervals), the prover has I geo-commitments $\{C_m^{\mathsf{geo}}\}_{m=1}^{I}$.

The $\mathsf{ComFrag}$ function, a crucial component of the soundness definition (Def. 1), in $\mathsf{GoAT\text{-}P}$ is specified now. Recall that $\mathsf{ComFrag}$ takes a file fragment as input and outputs a commitment of the same. The linear combination of file blocks $\boldsymbol{\mu}$ computed as part of $\mathsf{SW\text{-}P.Commit}$ is the file fragment, and the vector commitment is the $\mathsf{ComFrag}$ function, i.e., $\mathsf{ComFrag}(\boldsymbol{\mu}) = \mathsf{CPVC.Commit}(\boldsymbol{\mu})$. This implies that proving soundness of $\mathsf{GoAT\text{-}P}$ boils down to showing that the vector commitment over $\boldsymbol{\mu}$ is indeed computed inside the desired region.

It is worth emphasizing again why we had to modify the SW scheme in the first place. It might seem that our idea would also work with the original unmodified SW scheme. The prover could compute the full SW proof instead of commitment in step 2. But this would result in much worse geolocation accuracy due to the additional computational latency incurred in computing a full proof. The relation between computational latency and geolocation accuracy will be explained in the "Setting Δ" section.

PoRet computation phase ($\mathsf{PoRCompute}$*):* Once an epoch ends, the prover computes PoRets corresponding to the commitments computed during the epoch.

A naïve approach is to simply run the $\mathsf{SW.Prove}$ function I times with the same challenge sets used in step 2 of the geo-commit phase. But this creates large proofs. (Looking ahead, $\mathsf{GoAT\text{-}H}$ takes this naïve approach.)

Instead we aggregate proofs in much the same way as $\mathsf{SW.Prove}$, except for one key step, *coefficient randomization*. We derive a set of pseudorandom coefficients $\{r_j\}$ from the final PoRet commitment com_I. Denote the challenge set used to compute the j^{th} PoRet commitment by $\mathcal{S}_j = \{c_{ij},v_{ij}\}_{i=1}^{k}$ where $j \in \{1,...,I\}$. The newly generated coefficients are incorporated into those for the challenge sets, $\forall j, \mathcal{S}_j^* = \{c_{ij},r_j v_{ij}\}_{i=1}^{k}$. The modified challenge sets are aggregated as $\mathcal{S}^* = \cup_{j=1}^{I} \mathcal{S}_j^*$. We provide the rationale behind this technique in the Appendix (App. G) due to lack of space.

Given $\mathcal{S}^*$, the PoRet is computed as $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{SW.Prove}(\eta,\mathcal{S}^*)$.

The full proof of geo-retrievability then consists of the I geo-commitments and the PoRet, $\pi^{\mathsf{geo}} = \left\{ \{C_m^{\mathsf{geo}}\}_{m=1}^{I},\ \pi^{\mathsf{PoRet}} \right\}$. π^{geo} is given to V for verification.

Proof verification (Verify)**:** The verifier checks anchor transcripts in the I geo-commitments using the anchor's public key. Then the verifier derives PoRet challenges from transcript signatures as in proof generation. The coefficients $\{r_j\}$ and aggregate challenge set $\mathcal{S}^*$ are similarly computed. V computes an aggregate commitment $C^* = \prod_{j=1}^{I} (\mathsf{com}_j)^{r_j}$. The proof of retrievability π^{PoRet} and C^* are verified by $\mathsf{SW.Verify}(pp, \mathcal{S}^*, C^*, \pi^{\mathsf{PoRet}})$. Note that verification succeeds even with randomization of the challenge coefficients because $\mathsf{SW\text{-}P.Commit}$ contains only linear operations and the VC is homomorphic.

The final verification step is to check that the two timestamps are close in all geo-commitments, namely that $timer = t_2 - t_1$ and $timer \leq \Delta(L_{\mathsf{A}}, L_{\mathsf{P}})$, where Δ is a pre-agreed upon function that takes anchor and prover locations as inputs and outputs the maximum permissible run time of $\mathsf{GeoCommit}$ operations (those happening between times t_1, t_2).

Setting Δ: Choosing Δ involves striking a balance between completeness and soundness. To achieve completeness, Δ should output high enough values for honest parties to succeed. At the same time, Δ should output low enough values to prevent cheating, i.e., improper location of a file, by a cheating prover.

As shown in Fig. 2, the time difference captures the time taken to run two operations: a $\mathsf{GetAuthTime}$ API call and a PoRet commitment. Denote the maximum time for the two operations by t_{ping} and t_{com} respectively. We then have $\Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) = t_{\mathsf{ping}} + t_{\mathsf{com}}$.

Elapsed time for the $\mathsf{GetAuthTime}$ API call depends on the physical distance between the anchor and prover. We have $t_{\mathsf{ping}} = \mathsf{rtt}_{\mathsf{max}}(L_{\mathsf{A}}, L_{\mathsf{P}}) + t_{\mathsf{proc}}$, where the first term denotes the maximum round trip time introduced in Sec. 3.2 and t_{proc} denotes the maximum processing time by the prover and anchor (processing time accounts for the time taken to compute a response, see Sec. 5.1 for details). Therefore we have:

$$\Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) = (2 \cdot \mathsf{dist}(L_{\mathsf{A}}, L_{\mathsf{P}}) / S_{\mathsf{min}}) + t_{\mathsf{setup}} + t_{\mathsf{proc}} + t_{\mathsf{com}}. \quad (1)$$

The radius of the region of uncertainty (ROU) is $\delta = \Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \cdot S_{\mathsf{max}}/2$, i.e., for an adversarial prover to succeed in a PoGeoRet proof for F, most of F must be stored within δ distance of the anchor location L_{A}. This directly implies that smaller Δ results in a smaller δ, i.e., a better geolocation guarantee.

Out-sized effect on geolocation: We illustrate the out-sized effect some parameters have on the geolocation guarantee. Each millisecond added to t_{com} causes an equal increase in Δ due to eq. (1), and consequently a $S_{\mathsf{max}}/2$ increase in the ROU radius. Setting $S_{\mathsf{max}} = \frac{2}{3}c$ from Sec. 5, we see that $S_{\mathsf{max}}/2 \approx 10^5$m, i.e., a 100km reduction in geolocation accuracy. A similar effect is seen due to t_{setup} (connection setup cost) and t_{proc} (processing time).

Further considerations like grinding attacks (malicious prover repeatedly pinging an anchor to get a favorable challenge) are discussed in App. G.

4.2 GoAT extensions

Low-resolution anchors (TLS 1.2): Recall that $\mathsf{GeoCommit}$ assumes high-time-resolution anchors. But most existing anchors today (e..g, TLS 1.2 servers) only offer one-second time granularity. To address this, we introduce a technique we call *amplification*. The idea is to chain a sequence of proofs. Specifically, the prover alternates between computing a PoRet commitment and pinging the anchor, filling an entire resolution tick this way. The length of the proof chain is set by the *amplification factor a*. App. E contains more details. In brief, we show that amplification causes only a minor degradation in geolocation quality.

Decentralizing trust among anchors: It is straightforward to consider an extension to GoAT where as long as a threshold t number of anchors do not collude, the system is secure. The prover now has to invoke Prove once per anchor. The extra computational burden on the prover is minimal because proofs can be executed in parallel and the basic computation costs are negligible. The geolocation quality degrades due to the use of multiple anchors. Previously each anchor produced a circular ROU centered at its location, but with $t+1$ anchors, the new ROU is the union of the $t+1$ spherical circles as some t of them might be corrupt.

5 Implementation and Evaluation

We implemented GoAT-P in $\approx$2500 lines of C with support for both TLS 1.2 and Roughtime anchors. We use TLSe [40] for TLS, Roughenough [39] for Roughtime, and Relic [10] for pairings. GeoCommit is optimized using the asynchronous I/O library libaio [4]. We present our evaluation results in Sec. 5.1.

We perform a small experiment over a week using 10 anchors (5 in the US and 5 in the UK) to validate GoAT's performance in the wild. The experiment details and results are in Sec. 5.2.

Setup considerations: For the purposes of this paper, we only aim to demonstrate feasibility. We thus set parameters conservatively, favoring strong completeness with somewhat coarser geolocation than may be achievable in practice.

We set the maximum network speed of an adversary $S_{\mathsf{max}} = \frac{2}{3}c$ where c is the speed of light. This is the max. speed achievable in a fiber-optic cable [30].

Estimating the minimum speed for an honest user S_{min} can be tricky due to inconsistent network quality across locations. Based on RTT data from Wonder Network [37], we set $S_{\mathsf{min}} = \frac{2}{9}c$, i.e., advantage $\omega = S_{\mathsf{max}}/S_{\mathsf{min}} = 3$ and the constant startup cost $t_{\mathsf{setup}} = 5$ms. These parameter choices are consistent with recent work [17] that estimates the median RTT between PlanetLab nodes and popular websites to be about 3.2$\times$ slower than speed of light; so $S_{\mathsf{min}} = \frac{c}{4.5}$ is conservative. These parameters worked consistently across our experiments, and we emphasize again that our flexible-challenge model permits a prover to make multiple proof attempts over a given interval, creating strong resilience to network fluctuations.

Existing anchor discovery: To show that there is an existing network of servers that can serve as GoAT anchors, we perform a limited measurement study of existing TLS and Roughtime servers. We obtain server locations from the IP geolocation database, IP2Location.[5]

For TLS 1.2, we focus on domains for educational institutions, as we find they are more likely than other domains to have unique physical locations. We take the first 2850 domains from the Alexa top 1M list [1] containing the substring ".edu". We retain only those servers that return the correct time and whose ISP does not

[5] These databases are known to have some errors [24] and a rigorous geolocation experiment like [42] would have to be done before deploying GoAT.

belong to a CDN provider (as they do not have a unique location). The result is a set of 300 domains that can be used as anchors, i.e., 10.5% of our original list.

We are aware of four Roughtime servers as of Apr. 2023. All of them return correct time with microsecond granularity. To check that their locations are unique, we sent an ICMP ping request from two vantage points: North Virginia (NV) and Singapore (SP). In this process, we identified one of the servers as unusable for geolocation, as it has a small RTT from both NV and SP (17ms and 30ms respectively), suggesting the use of a CDN.

GoAT parameters: For SW PoRet, we use the BLS12-381 curve. Except in one experiment below, we set the number of sectors per block, $s = 96$. For GoAT to be secure, we need $(\rho + \phi + (1 - 2^{-(\lambda+\alpha)/l}))^l$ to be negligible (App. C). Assuming the grinding constraint $\alpha = 40$, one set of parameters to achieve 128-bit security are code rate $\rho = 0.33$ and number of challenges $l = 250$; note that the bandwidth overhead is set to $\phi = 0.001$ (based on App. A.3). For our experiments, we set the number of challenges to $l = 100$. We expect minimal impact on results due to the slightly lower l. Details on how we set anchor processing times can be found in App. G.2 and a discussion over parameter tradeoffs is in App. G.

5.1 Evaluation

We evaluate GoAT through several benchmarks on an AWS c5.4xlarge machine with 16 CPU, 32GB RAM and 2TB io2 SSD that is capped at 20k IOPS. We always take 50 samples to compute the mean and standard deviation (in brackets).

PoRet commit time (vs) file size: As explained in Sec. 4, PoRet commit time has a direct impact on the ROU radius. Table 1 presents the time taken to compute the PoRet commitment as a function of file size (128MB to 256GB). The times are all small (1-4ms) thanks to our parallelized implementation (we set $t_{\mathsf{com}} = 2$ms which works for files below 16GB). Of the numbers shown, about 1ms is spent on the actual commitment computation, while the rest is for file reads. We use x64 Assembly accelerated code provided by Relic for EC operations and further optimize it using a multi-threaded implementation (by breaking up a vector into smaller ones). The file read times are largely constant except for an abrupt jump at 64GB. This happens because the cache is no longer useful and therefore we switch to using Direct I/O.

Computation costs: Table 1 presents the time taken for the Prove and Verify operations. Here we assume a fixed epoch length and vary the number of intervals. Recall that with more intervals per epoch, the location guarantee gets better. As shown, with 100 intervals, Prove takes about 2s and Verify takes around 3s. Concrete costs are negligible for both operations (our AWS instance cost us $0.376 per hour). Also note that the effect of number of intervals on both Prove and Verify computation times is close to linear. We set the amplification factor a to 1. A similarly linear effect is expected if a is varied.

Communication costs: With Roughtime anchors and the BLS12-381 curve, GoAT-P proof size is $1941 + 720I$ bytes. The constant part of the equation is

File size	Time (ms)
128MB	1.09 (0.02)
1GB	1.02 (0.02)
4GB	1.02 (0.02)
16GB	1.04 (0.02)
64GB	4.27 (0.22)
256GB	4.06 (0.22)

#intervals	PoRCompute	Verify
1	18.11 (0.06)	47.28 (0.02)
10	183.65 (0.43)	320.81 (5.59)
100	1,838.44 (0.73)	2,991.14 (61.03)

Table 1: (Left) Time taken for SW-P.Commit. (Right) Computation time of PoRCompute and Verify in ms (vs) no. of intervals per epoch. Standard deviations are in brackets.

due to the communication-efficient PoRet proofs, while the rest by anchor transcripts. If $I = 100$, GoAT-P proof size is 72.2KB with a dominating 70KB of anchor transcripts. Note in particular how the PoRet component of the proof size does not change with the number of intervals.

The proofs are larger with TLS anchors due to amplification. For example, the proof size of GoAT-P with a TLS anchor of $a=20$ is 799.64KB.

GoAT-H also increases the proof sizes. For example, proofs are around 4.5MB with a TLS anchor (about 5x bigger than GoAT-P) and 265.5KB with a Roughtime anchor (about 3x bigger). See Fig. 13 for proof sizes for all GoAT variants.

5.2 Real-world experiment

We devise a small experiment to demonstrate the practical feasibility of GoAT, specifically how it deals with network volatility. We focus on the GeoCommit protocol alone as it is the sole operation affected by network conditions. Prior works [27] have observed network stability over long time periods, and conclude that network instability is frequent but most often transient. So we handle failures in GeoCommit by simply retrying until success. Concretely, the number of retries is capped at 30 with a gap of 1 second between retries. In this process, we count the number of retries needed to succeed and the false rejection rate, if any. Under ideal network conditions, 0 retries are expected.

We run the prover from two AWS instances located in North Virginia (NV) and London (LON).[6] Five anchors, screened for the criteria described before, are picked near each. The interval length is set to $\beta=30$mins and the GeoCommit protocol is run for 10 days at NV (525 intervals) and 7 days at LON (347 intervals).

Table 2 shows the ten anchors used including three Roughtime anchors. The first five anchors are used with the AWS instance in NV, while the remaining with the one in LON. The 2nd column shows the distance d between the anchor and its corresponding AWS instance, the 3rd column shows the amplification factor a, and the 4th column shows the ROU radius δ along with the ratio δ/d.

Geolocation accuracy: Recall that the ROU radius is given by $\delta = d\omega + (t_{\mathsf{com}} + t_{\mathsf{setup}} + t_{\mathsf{proc}}) \cdot S_{\max}/2$ (eq. (1)). Table 2 shows that δ/d converges to the advantage $\omega = 3$ for storage providers farther away from the anchor, suggesting that

[6] We use a 100 IOPS, 30GB gpt2 SSD instead of io2 SSD as the latter is more expensive. We did not find a significant impact on commit times due to this.

distance-to-the-anchor is the dominating factor in determining the geolocation accuracy. But for nearby providers, we see high ratios going up to 46 caused by constants like t_{setup} and t_{proc}. Also note that TLS anchors achieve worse geolocation compared to Roughtime ones due to the higher processing times; for example, compare "american.edu" and "roughtime.chainpoint.org".

If finer geolocation is desired, aggressive parametrization can help. For example, the variable t_{setup} (startup cost) alone is responsible for nearly half the geolocation radius of "roughtime.chainpoint.org". It can be reduced with a refined network model, as we found that only some anchors require this extra time (App. G).

Anchor name	Distance	a	ROU radius (δ/d)	#retries (SD)
roughtime.chainpoint.org	46.00	1	1187.27 (25.81)	0.06 (0.23)
roughtime.sandbox.google.com	115.33	1	1395.26 (12.10)	0.02 (0.13)
www.american.edu	43.99	60	1665.51 (37.86)	0.03 (0.47)
www.sunysuffolk.edu	450.29	34	2939.14 (6.53)	1.00 (0.06)
roughtime.int08h.com	1582.83	1	5797.76 (3.66)	0.01 (0.11)
holycross.ac.uk	35.26	61	1638.21 (46.46)	0 (0)
sruc.ac.uk	58.83	58	1722.94 (29.29)	0.67 (1.04)
gold.ac.uk	87.45	55	1816.92 (20.78)	1.02 (0.15)
nott.ac.uk	175.19	48	2081.89 (11.88)	2.26 (1.76)
www.ed.ac.uk	533.67	31	3223.57 (6.04)	0.003 (0.05)

Table 2: The second column shows the distance between the anchor and its closest AWS instance (d). The third column shows the amplification factor (a). The fourth column shows both the ROU radius (δ) and the ratio (δ/d). All distances are in km. Last column shows the mean and SD of the number of retries.

6 Conclusion

We presented GoAT, a practical Proof of Geo-Retrievability (PoGeoRet) scheme for file geolocation. GoAT leverages timestamping internet servers for proving location and the Shacham-Waters PoRet scheme for proving file retrievability. GoAT has a unique challenge model that permits batching proofs over several intervals and verifsying them at the end of an epoch; this also makes GoAT proofs small. We demonstrated GoAT's practicality through a fully functional implementation and a real-world experiment.

Finally note that while GoAT can be used to construct a Proof of Replication (PoRep), GoAT cannot be used as a like-for-like replacement of existing PoReps [22] as PoReps are often designed to be mining functions. Extending GoAT for mining is an open problem.

7 Acknowledgments

We would like to thank Filecoin researchers, in particular Nicola Greco and Will Scott, for thoughtful discussions that helped shape the work.

This work was funded by NSF grants CNS-1564102, CNS-1704615, and CNS-1933655 as well as generous support from IC3 industry partners. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and may not reflect those of these sponsors.

References

1. Alexa top sites. https://www.alexa.com/topsites, [Accessed Apr 2021]
2. Coinmarketcap, cryptocurrency market prices, https://coinmarketcap.com/, [Accessed Aug 2022]
3. Filecoin aims to use blockchain to make decentralized storage resilient and hard to censor (2021), https://www.infoq.com/news/2021/02/filecoin-blockchain-storage/, [Accessed Jul 2022]
4. Linux-native asynchronous i/o access library. https://pagure.io/libaio (2021), [Accessed Jul 2022]
5. Metric space. https://en.wikipedia.org/wiki/Metric_space (2021), [Accessed Jul 2022]
6. Ssd userbenchmarks - 1058 solid state drives compared. https://ssd.userbenchmark.com/ (2021), [Accessed Apr 2021]
7. A. Malhotra, A.L., Ladd, W.: Roughtime. https://datatracker.ietf.org/doc/html/draft-roughtime-aanchal (2020)
8. Amazon: Aws ec2 costs. https://aws.amazon.com/ec2/pricing/on-demand/ (2021), [Accessed Apr 2021]
9. Amazon: Aws s3. https://aws.amazon.com/s3/ (2021)
10. Aranha, D.F., Gouvêa, C.P.L., Markmann, T., Wahby, R.S., Liao, K.: RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic
11. Ateniese, G., Burns, R., Curtmola, R., Herring, J., Kissner, L., Peterson, Z., Song, D.: Provable data possession at untrusted stores. In: ACM CCS. pp. 598–609 (2007)
12. Bellare, M., Palacio, A.: The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In: Annual International Cryptology Conference. pp. 273–289. Springer (2004)
13. Benet, J.: IPFS - content addressed, versioned, P2P file system. CoRR (2014), http://arxiv.org/abs/1407.3561
14. Benet, J., Greco, N.: Filecoin: A decentralized storage network. Protoc. Labs pp. 1–36 (2018)
15. Benet, J., Dalrymple, D., Greco, N.: Proof of replication. Protocol Labs, July p. 20 (2017)
16. Benson, K., Dowsley, R., Shacham, H.: Do you know where your cloud files are? In: Proceedings of the 3rd ACM workshop on Cloud computing security workshop. pp. 73–82 (2011)
17. Bozkurt, I.N., Aguirre, A., Chandrasekaran, B., Godfrey, P.B., Laughlin, G., Maggs, B., Singla, A.: Why is the internet so slow?! In: International Conference on Passive and Active Network Measurement. pp. 173–187. Springer (2017)
18. Cecchetti, E., Fisch, B., Miers, I., Juels, A.: Pies: Public incompressible encodings for decentralized storage. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 1351–1367 (2019)
19. Clark, M.: Nfts, explained (Mar 11, 2021), https://www.theverge.com/22310188/nft-explainer-what-is-blockchain-crypto-art-faq, [Accessed Jul 2022]

20. Dierks, T., Rescorla, E.: TLS 1.2 RFC 5246. https://tools.ietf.org/html/rfc5246 (2008)
21. of Entropy, L.: Drand: Distributed randomness beacon. drand.love (Accessed Aug 2022)
22. Fisch, B.: Poreps: Proofs of space on useful data. IACR Cryptol. ePrint Arch. **2018**, 678 (2018)
23. Fisch, B., Bonneau, J., Greco, N., Benet, J.: Scaling proof-of-replication for filecoin mining. Benet//Technical report, Stanford University (2018)
24. Gill, P., Ganjali, Y., Wong, B., Lie, D.: Dude, where's that ip? circumventing measurement-based ip geolocation. In: Proceedings of the 19th USENIX conference on Security. pp. 16–16 (2010)
25. Hanser, C., Slamanig, D.: Efficient simultaneous privately and publicly verifiable robust provable data possession from elliptic curves. In: 2013 International Conference on Security and Cryptography (SECRYPT). pp. 1–12. IEEE (2013)
26. Harding, E.L., Acevedo, L.J., Dailey, L.R.: Data localization and data transfer restrictions. https://www.natlawreview.com/article/data-localization-and-data-transfer-restrictions/ (August 2021), [Accessed Jul 2022]
27. Høiland-Jørgensen, T., Ahlgren, B., Hurtig, P., Brunstrom, A.: Measuring latency variation in the internet. In: Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies. pp. 473–480 (2016)
28. Jeon, K.E., She, J., Soonsawad, P., Ng, P.C.: Ble beacons for internet of things applications: Survey, challenges, and opportunities. IEEE Internet of Things Journal (2018). https://doi.org/10.1109/JIOT.2017.2788449
29. Juels, A., Kaliski Jr, B.S.: Pors: Proofs of retrievability for large files. In: Proceedings of the 14th ACM conference on Computer and communications security. pp. 584–597 (2007)
30. Katz-Bassett, E., John, J.P., Krishnamurthy, A., Wetherall, D., Anderson, T., Chawathe, Y.: Towards ip geolocation using delay and topology measurements. In: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. pp. 71–84 (2006)
31. Kohls, K., Diaz, C.: VerLoc: Verifiable localization in decentralized systems. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2637–2654. USENIX Association, Boston, MA (Aug 2022), https://www.usenix.org/conference/usenixsecurity22/presentation/kohls
32. Labs, P.: Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf (July 19, 2017), [Accessed Jul 2022]
33. Labs, S.: Storj: A decentralized cloud storage network framework. https://www.storj.io/storjv3.pdf (October 30, 2018), [Accessed Jul 2022]
34. Mellor, C.: Ssds will crush hard drives in the enterprise, bearing down the full weight of wright's law. https://blocksandfiles.com/2021/01/25/wikibon-ssds-vs-hard-drives-wrights-law/ (January 25, 2021), [Accessed Jul 2022]
35. Patton, C.: Roughtime: Securing time with digital signatures. https://blog.cloudflare.com/roughtime/ (2018), [Accessed Jul 2022]
36. Qualys: Ssl pulse. https://www.ssllabs.com/ssl-pulse/ (2022), [Accessed Jul 2022]
37. Reinheimer, P., Roberts, W.: Global ping statistics: Manhattan. https://wondernetwork.com/pings/Manhattan, [Accessed Apr 2021]

38. Shacham, H., Waters, B.: Compact proofs of retrievability. In: International conference on the theory and application of cryptology and information security. pp. 90–107. Springer (2008)
39. Stock, S.: Roughenough. https://github.com/int08h/roughenough (2021), [Accessed Jul 2022]
40. Suica, E.: Single c file tls 1.2/1.3 implementation. https://github.com/eduardsui/tlse/ (2021), [Accessed Jul 2022]
41. Vorick, D., Champine, L.: Sia: Simple decentralized storage. https://sia.tech/sia.pdf (November 29, 2014), [Accessed Jul 2022]
42. Wang, Y., Burgener, D., Flores, M., Kuzmanovic, A., Huang, C.: Towards street-level client-independent ip geolocation. In: NSDI. vol. 11, pp. 27–27 (2011)
43. Watson, G.J., Safavi-Naini, R., Alimomeni, M., Locasto, M.E., Narayan, S.: Lost: location based storage. In: Proceedings of the 2012 ACM Workshop on Cloud computing security workshop. pp. 59–70 (2012)

A Formalism details

Storage devices: To model an adversary that can place files in several distinct locations, we introduce a model for *(storage) devices*. We denote a device by D. In our security experiments (for soundness), all devices are under the control of the adversary. The adversary can place devices in locations of its choice but those locations remain fixed throughout the experiment. Devices have access to unlimited storage memory. The adversary *cannot* execute any function outside the device environment, a requirement that simplifies our model w.l.o.g., as the adversary can transmit files freely between storage devices placed in locations of its choice. Formally, we model all devices by way of an oracle $\mathcal{O}_{\mathsf{dev}}$ presented in App. A.1.

Modeling time: As noted before, in all PoGeoRet schemes, the verifier V uses an internal clock to time interactions with the prover P. V must also know an upper bound on the computation time required by an honest P.
We allow the adversary to communicate messages (of any size) between devices with speed S_{max}.

A.1 Soundness

Our security definition for soundness has many similarities to that of a PoRet. The PoRet soundness definition involves two experiments: *setup* and *challenge*. The setup experiment lets the adversary set up its devices and pick a file F for the challenge-response interactions in the challenge experiment. The challenge experiment corresponds to interactions with a real-world verifier, and requires that an adversary responds to ϵ-fraction of queries correctly. The challenge experiment interface is reused for extraction, in which a verifier tries to reconstruct F from file fragments obtained in the $\mathsf{Extract}$ protocol. A PoRet scheme is said to be sound if success in the challenge experiment implies that extraction succeeds.

The PoGeoRet soundness definition includes these requirements, but *also that success means the file F is inside the region R^{rou}*. To capture this requirement, we introduce into PoGeoRet a *commitment oracle* $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ that models the

ComFrag function in a localized way. At a high level, $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ models the necessity of computing ComFrag in a real-world execution similar to how a random oracle models that of a hash function. In particular, $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ models running ComFrag on a storage device inside R^{rou} and consequently captures the location where the file fragment $\boldsymbol{\mu}$ input to ComFrag is stored. Our modeling of a commitment oracle is similar to how a random oracle models a hash function, but generalized to a generic ComFrag function and localized to a geographic region. (In one of the GoAT protocols, ComFrag is a hash function, and the commitment oracle then becomes a localized random oracle.) Note that the adversary can run ComFrag on any device it wants, either inside or outside R^{rou}, but $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ only tracks executions inside R^{rou}. $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ is a subroutine of the device oracle $\mathcal{O}_{\mathsf{dev}}$.

Extraction, now called "geo-extraction," is deemed successful only if F can be computed from a set of file fragments μ^{all} such that every fragment $\boldsymbol{\mu} \in \mu^{\mathsf{all}}$ was previously seen in a query to $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$. The idea is that, if $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ was queried on a fragment, it happened inside R^{rou}. Consequently, if enough file fragments are inside R^{rou}, then the file F itself is in R^{rou}. The rest of the PoGeoRet soundness definition is same as for a PoRet, as explained in more detail below.

Corresponding to the setup and challenge experiments, the adversary $\mathcal{A}$ consists of two parts, $\mathcal{A}_{\mathsf{setup}}$ and $\mathcal{A}_{\mathsf{chal}}$, each involved only in its respective experiment.

$\mathcal{A}_{\mathsf{setup}}$ may interact arbitrarily with the verifier; it may create files and run Setup on them; it may also undertake challenge-response interactions with the verifier and observe if the verifier accepts or not. $\mathcal{A}_{\mathsf{setup}}$ is allowed to place any number of devices at locations of its choice and decide what to store in their memories. Device locations are fixed after creation.

The setup experiment runs Setup on a file F picked by the adversary. The resulting output F^* is challenged in the challenge experiment.

During the challenge experiment, challenges are issued to the second adversary component $\mathcal{A}_{\mathsf{chal}}$ and success is based on whether the proof verifies.

Geo-extraction is the crux of the soundness definition. Just as a proof of knowledge has knowledge-soundness if success with the verifier implies an ability to extract a witness, a PoGeoRet is sound only if success with the verifier implies an ability to extract the target file from a target geographical region—which, again, we refer to as "geo-extraction."

Geo-extraction consists of three steps. First Extr.Derive derives file fragments μ^{all} from interactions with $\mathcal{A}_{\mathsf{chal}}$, i.e., the same adversary component as in the challenge experiment. We allow the adversary to be rewound in this step, as is standard in the PoRet literature, e.g., [29,38]. Second, Extr.Assemble tries to recompute the file from the derived file fragments μ^{all}. Extr.Assemble does not interact with the adversary. The third and final step is verifying if all the fragments μ^{all} were seen inside R^{rou}, i.e., as inputs to the commitment oracle $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$. We say that geo-extraction succeeds only if *both* Extr.Assemble succeeds *and* all file fragments were in R^{rou}. A PoGeoRet scheme is said to be sound if adversarial success in the challenge experiment implies that geo-extraction succeeds w.h.p.

From the point of view of an adversary whose goal is to "cheat" a verifier, $\mathcal{A}$ wants to create an environment in which V believes the file is in R^{rou}, but it isn't. Thus the aim of $\mathcal{A}_{\mathsf{setup}}$ is to set up devices in such a way that: (1) V accepts responses from $\mathcal{A}_{\mathsf{chal}}$ in the challenge experiment and (2) V cannot geo-extract the file F, i.e., fails to recompute F from the file fragments input to $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$.

We now present the device oracle formally in App. A.1. Our detailed security experiments and soundness definition are in App. A.1.

Device oracle We model device actions in our experiments via the device oracle $\mathcal{O}_{\mathsf{dev}}$ specified in Fig. 3.

- $\mathcal{O}_{\mathsf{dev}}.\mathsf{init}$ is used in the setup experiment for initialization.
- $\mathcal{O}_{\mathsf{dev}}.\mathsf{createDevice}$ allows the adversary to spawn devices at any location (both inside and outside R^{rou}) and $\mathcal{O}_{\mathsf{dev}}.\mathsf{setStorage}$ allows changes to device storage.
- $\mathcal{O}_{\mathsf{dev}}.\mathsf{exec}$ allows the adversary to execute a function func on a device of its choice, including any function in the PoGeoRet API except $\mathsf{ComFrag}$. $\mathcal{O}_{\mathsf{dev}}.\mathsf{execComFrag}$ models the commitment oracle $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ and tracks all calls to $\mathsf{ComFrag}$. $\mathcal{O}_{\mathsf{dev}}.\mathsf{exec}$ can also communicate with other devices through $\mathcal{O}_{\mathsf{dev}}.\mathsf{sendTo}$ or execute code on a different device.
- $\mathcal{O}_{\mathsf{dev}}.\mathsf{seenInROU}$ is used only in the soundness definition to check if a given set of inputs were previously seen in a call to $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$.

Device oracle $\mathcal{O}_{\mathsf{dev}}$

```
1: State: A region $R^{\mathsf{rou}}$. Key-value pairs $\mathcal{D}[did] = (loc, s)$ where the key $did$ is the device identifier, $loc$ is its location, $s$ is an unbounded list representing device storage. A list $\mu^{\mathsf{rec}}$ to track calls to ComFrag made from within $R^{\mathsf{rou}}$.
2: init($R$): Set $R^{\mathsf{rou}} = R$. Not callable by the adversary.
3: createDevice($loc, s$): Create a unique device identifier $did$ and set $\mathcal{D}[did] = (loc, s)$. Return $did$.
4: exec($did$, func) $\to out$: If $did \notin \mathcal{D}$ return $\perp$. Compute any function func $\neq$ ComFrag and return its output. func can read / write to $\mathcal{D}[did].s$ or call any $\mathcal{O}_{\mathsf{dev}}$ functions internally.
5: execComFrag($did, \boldsymbol{\mu}$): If $did \notin \mathcal{D}$ return $\perp$. Else compute ComFrag($\boldsymbol{\mu}$) and return its output. If $\mathcal{D}[did].loc \in R^{\mathsf{rou}}$, do $\mu^{\mathsf{rec}}$.append($\boldsymbol{\mu}$).
6: sendTo($did_1, did_2, data$): If $data \in \mathcal{D}[did_1].s$, $\mathcal{D}[did_2].s$.append($data$).
7: setStorage($did, s_{new}$): Modify the device storage to $\mathcal{D}[did].s = s_{new}$.
8: seenInROU($\mu^{\mathsf{all}}$): Return 1 if $\forall \boldsymbol{\mu} \in \mu^{\mathsf{all}}, \boldsymbol{\mu} \in \mu^{\mathsf{rec}}$ holds.
```

Fig. 3: The device API.

Soundness experiments and definition In all experiments, the adversary has complete freedom to call any device function. Both the experiments are in Fig. 4.

```
$\mathbf{Exp}^{\mathsf{setup}}_{\mathcal{A}}(R)$
$\mathcal{O}_{\mathsf{dev}}.\mathsf{init}(R)$
$(sk,pk) \leftarrow \mathsf{KGen}(1^\lambda)$
$F \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{dev}}}_{\mathsf{setup}}(pk)$
$(F^*,\eta,pp) \leftarrow \mathsf{Setup}(sk,pk,F)$
$s \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{dev}}}_{\mathsf{setup}}(F^*,\eta,pp)$
return $(\eta,pp,F,s)$
```

```
$\mathbf{Exp}^{\mathsf{chal}}_{\mathcal{A}}(\eta,R,pp,s)$
$\mathcal{A}^{\mathcal{O}_{\mathsf{dev}}}_{\mathsf{chal}}(s)$   % Init $\mathcal{A}_{\mathsf{chal}}$
$c \leftarrow \mathsf{Chal}(\eta,pp)$   % Random chal
$timer,\pi^{\mathsf{geo}} \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{dev}}}_{\mathsf{chal}}.\mathsf{Prove}(\eta,R,c)$
return $\mathsf{Verify}(pp,R,c,\pi^{\mathsf{geo}},timer)$
```

Fig. 4: Setup and Challenge Experiments.

In the setup experiment $\mathbf{Exp}^{\mathsf{setup}}_{\mathcal{A}}$, Setup is run over a file F and the output given to $\mathcal{A}$, who decides where to place the file. $\mathcal{A}_{\mathsf{setup}}$ outputs state s that is given to $\mathcal{A}_{\mathsf{chal}}$ as initial input.

In the challenge experiment $\mathbf{Exp}^{\mathsf{chal}}_{\mathcal{A}}$, $\mathcal{A}_{\mathsf{chal}}$ responds to a challenge issued by the verifier. Note that we issue one PoGeoRet challenge which internally comprises one PoRet challenge. The success probability for the challenge experiment is defined as:

$$\mathbf{Succ}^{\mathsf{cha}}_{\mathcal{A}}(\eta,R,pp,s) = \Pr\Big[\mathbf{Exp}^{\mathsf{chal}}_{\mathcal{A}}(\eta,R,pp,s) = 1\Big].$$

Definition 1 (Soundness). *A PoGeoRet scheme is (ϵ,p)-sound w.r.t. a target region R^{target} achieving a region of uncertainty R^{rou} if for all poly-time $\mathcal{A}$:*

$$\Pr\left[\begin{array}{c|c} \mu^{\mathsf{all}} \leftarrow \mathsf{Extr.Derive}(\eta,R^{\mathsf{rou}},pp), & (\eta,pp,F,s) \leftarrow \mathbf{Exp}^{\mathsf{setup}}_{\mathcal{A}}(R^{\mathsf{rou}}), \\ \mathsf{Extr.Assemble}(\mu^{\mathsf{all}}) = F, & \mathbf{Succ}^{\mathsf{cha}}_{\mathcal{A}}(\eta,R^{\mathsf{rou}},pp,s) \geq \epsilon \\ \mathcal{O}_{\mathsf{dev}}.\mathsf{seenInROU}(\mu^{\mathsf{all}}) = 1 & \end{array}\right] \geq p.$$

This definition states that a PoGeoRet scheme is (ϵ,p)-sound if, for every adversary that succeeds the challenge experiment with ϵ probability, geo-extraction must also succeed with probability p. Sometimes we omit p and say that a PoGeoRet scheme is ϵ-sound; in such cases we mean that p is negligibly close to 1, i.e., $p = 1 - \mathsf{negl}(\lambda)$.

A.2 Completeness

Completeness requires that a valid prover P using a device inside the target region R^{target} can successfully prove operation inside R^{rou}.

```
Experiment $\mathbf{Exp}^{\mathsf{comp}}(L,R^{\mathsf{rou}})$
$(sk,pk) \leftarrow\$ \mathsf{KGen}(1^\lambda), F \leftarrow\$ \{0,1\}^*$
$(F^*,\eta,pp) \leftarrow \mathsf{Setup}(sk,pk,F)$
$did \leftarrow \mathcal{O}_{\mathsf{dev}}.\mathsf{createDevice}(L,F^*)$
$c \leftarrow \mathsf{Chal}(\eta,pp)$
$timer,\pi^{\mathsf{geo}} \leftarrow \mathcal{O}_{\mathsf{dev}}.\mathsf{exec}(did,\mathsf{Prove}(\eta,R^{\mathsf{rou}},c))$
return $\mathsf{Verify}(pp,R^{\mathsf{rou}},c,\pi^{\mathsf{geo}},timer)$
```

Definition 2 (Completeness). *A PoGeoRet scheme is complete w.r.t a target region R^{target} achieving a region of uncertainty R^{rou}, if for any $L \in R^{\mathsf{target}}$,* $\Pr[\mathbf{Exp}^{\mathsf{comp}}(L,R^{\mathsf{rou}})=1] > 1-\mathsf{negl}(\lambda)$.

Note that we use the device oracle $\mathcal{O}_{\mathsf{dev}}$ to spawn a device at the target location L and run Prove on this device.

Remark: For simplified presentation, the above definitions of soundness and completeness assume an interactive protocol between the prover and verifier. Our main goal, though, is for GoAT to operate non-interactively. Due to lack of space, we relegate non-interactive definitions to App. F.1 (they only require minor modifications).

A.3 Extension to flexible-challenge model

As before, the setup experiment has the adversary pick file F and initialize several devices. But then, I challenge experiments take place, one per interval. After the epoch (or) I intervals end, the challenge responses are verified. Geo-extraction takes place after that.

The device oracle $\mathcal{O}_{\mathsf{dev}}$ now maintains a record of the commitment oracle queries made in each interval; let μ_i^{rec} denote the list of fragments queried in the ith interval. In each interval, the adversary requests a challenge at a time of its choosing. After the epoch (or I intervals) ends, we extract the file I times by running Extract. Geo-extraction in the ith interval succeeds if the file can be assembled from the fragments in μ_i^{rec}. Soundness is defined in the same way as before except that we now require geo-extraction succeed in all I intervals.

Economic argument: Note that for short intervals, typically $\phi \ll 1$, as we now show in an example. Consider the bandwidth and storage costs currently charged by Amazon. (We take storage cost as a proxy for revenue.) Suppose we set the interval length $\beta = 30$mins. If an encoded file size is $|F^*| = 1$TB, then a prover's storage revenue is at most \$0.02 per interval on Amazon S3 [9]. On the other hand, AWS bandwidth costs start from \$20 per TB[7].So downloading 1GB would cost the same as the revenue obtained by storing 1TB. Therefore $\phi = 1/1000$.

B GoAT-H

GoAT-H version: The key difference in GoAT-H is the use of a hash function as the vector commitment. This results in larger proofs and extra computational steps in Prove and Verify. GeoCommit is the same as before except the change in the PoRet commitment function, i.e. SW-H.Commit instead of SW-P.Commit. PoRet computation (Prove) involves naïvely running SW.Prove I times because the aggregation tricks do not work anymore. If $\mathcal{S}_j$ denotes the jth set of challenges, compute $\pi_j^{\mathsf{PoRet}} \leftarrow \mathsf{SW.Prove}(\eta, \mathcal{S}_j)$; the final proof

[7] AWS bandwidth costs vary by region, ranging from \$20-\$100 per TB transferred [8]. S3 charges also vary by region, we use the maximum above.

is $\pi^{\mathsf{PoRet}} \leftarrow \{\pi_1^{\mathsf{PoRet}}, \pi_2^{\mathsf{PoRet}}, \ldots, \pi_I^{\mathsf{PoRet}}\}$. Accordingly, verification involves running SW-H.Verify I times. The proof sizes in GoAT-H are asymptotically the same as GoAT-P, but concretely about 3x larger. Geolocation accuracy remains the same.

A summary of both the GoAT protocols can be found in Figure 11.

C GoAT security

Both GoAT-H and GoAT-P achieve the same security but in different models. GoAT-H operates in the random oracle model and its security proof relies on the commonly used "knowledge of queries" technique. On the other hand, GoAT-P's security relies on a new assumption that we call the KEV Assumption (KEVA). The proof sketches for both protocols are in App. D.

KEVA$_s$ extends the commonly used KEA1 [12] for an s-sized vector of elements. In particular, if $s=1$ it reduces to KEA1. It states that if $\mathcal{A}$ takes two correlated sets of bases $(\mathbf{h}_1, \mathbf{h}_2 = \mathbf{h}_1^b)$ as input and outputs (c_1, c_2) s.t. $c_2 = c_1^b$, then there exists an extractor $\mathcal{E}_{\mathcal{A}}$ that can output a pre-image $\mathbf{x}$ s.t. the Pedersen commitment of $\mathbf{x}$ with $\mathbf{h}_1$ is c_1, i.e., $\mathbf{h}_1^{\mathbf{x}} = c_1$ while using the same inputs as before. This is saying that the only way of computing (c_1, c_2) is by picking a pre-image $\mathbf{x}$ and computing its Pedersen commitment.

Definition 3 (KEVA$_s$)**.** *Given any set of distinct bases $\boldsymbol{h}_1 \in \mathbb{G}^s$, for any PPT $\mathcal{A}$, there exists a PPT extractor $\mathcal{E}_{\mathcal{A}}$ s.t.*

$$\Pr\left[\begin{array}{c} \mathbf{x} \leftarrow \mathcal{E}_{\mathcal{A}}(\boldsymbol{h}_1, \boldsymbol{h}_2), \\ \boldsymbol{h}_1^{\mathbf{x}} = c_1 \end{array} \middle| \begin{array}{c} b \leftarrow\$ \mathbb{Z}_p, \boldsymbol{h}_2 = \boldsymbol{h}_1^b \\ (c_1, c_2) \leftarrow \mathcal{A}(\boldsymbol{h}_1, \boldsymbol{h}_2), c_2 = c_1^b \end{array}\right] > 1 - \mathsf{negl}(\lambda).$$

Say the target region is a single location, $R^{\mathsf{target}} = (L; 0)$. Then the region of uncertainty achieved by GoAT-H and GoAT-P is a circle centered at anchor's location with radius $\delta_L = \Delta(L_{\mathsf{A}}, L) \cdot S_{\mathsf{max}}/2$. In practice, the target region might have a small diameter, $R^{\mathsf{target}} = (L; \delta')$. As long as δ' is small, we can approximate and define the region of uncertainty as $R^{\mathsf{rou}} = (\mathsf{A}; \delta'')$ where $\delta'' = \max_{\{L' \in R^{\mathsf{target}}\}} \delta_{L'}$.

Theorem 1. *Let $w = \left(\rho + \phi + 1 - 2^{\frac{-\lambda-\alpha}{l}}\right)^l$. For any $\epsilon \leq 1$ s.t. $\epsilon - w$ is positive and non-negligible and* KEVA$_s$ *holds and that the CDH problem is hard in bilinear groups,* GoAT-P *is (ϵ, p)-sound at a target geographic region $R^{\mathsf{target}} = (L; \delta')$ achieving a geolocation guarantee of $R^{\mathsf{rou}} = (\mathsf{A}; \delta'')$ under the flexible challenge model and the random oracle model.*

D Security Proof Sketches

We now provide a proof sketch for Thm. 1. We primarily focus on GoAT-P with a Roughtime anchor adding notes about how the proof extends to GoAT-H (or) to TLS anchors where needed.

Recall that the GoAT-P proof π^{geo} consists of I geo-commitments and a PoRet proof. Each geo-commitment C^{geo} consists of $a+1$ anchor transcripts and all but the first transcript contain a PoRet commitment. In total, $N = Ia$ PoRet commitments are in a proof. Similarly, the $\mathsf{GoAT\text{-}H}$ proof consists of $N = Ia$ PoRet proofs and I geo-commitments.

We prove soundness of GoAT in four steps.

1. Prove that the N PoRet commitments and the PoRet proof(s) are correctly computed, i.e., PoRet verification ($\mathsf{PoRet.Verify}$) part of Verify must detect otherwise.
2. A combination of timing and knowledge based arguments to prove that the $\mathsf{ComFrag}$ operation is run on a device inside R^{in}, i.e., prove that all file fragments part of a correct proof must have been queried to the commitment oracle $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$.
3. Prove that the extraction algorithm can efficiently reconstruct ρ fraction of file blocks from the fragments in each of the I snapshots $\{S_i\}_{i=1}^{I}$.
4. Prove that the file can be reconstructed from any ρ fraction.

The proof for part 4 follows directly from the properties of a rate-ρ erasure code, so we do not expand on it further.

D.1 Part-two proof

For this part, we need to prove that the commitment oracle $\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$ receives all file fragments that are part of a correct PoRet commitment, proof. (The latter is guaranteed by the part-one proof provided later.)

We proceed in two steps. First we argue that the only way of computing a valid PoRet commitment is by computing $\mathsf{ComFrag}$ on valid file fragments. This relies on the KEV assumption (See Def. 3) for $\mathsf{GoAT\text{-}P}$ and the ROM for $\mathsf{GoAT\text{-}H}$. Next we argue that all calls to $\mathsf{ComFrag}$ must take place from within the desired target region R^{in}. This relies on a timing based argument. Overall, this proves that if correct PoRet commitments and proofs are computed, then the commitment oracle ($\mathcal{O}^{\mathsf{rou}}_{\mathsf{com}}$) records all the corresponding file fragments.

The proof for first step is as follows. Given a valid PoRet commitment $C_{\boldsymbol{\mu}}$ for SW-P, we need to prove the existence of a valid pre-image $\boldsymbol{\mu}$. But the KEVA directly offers this. We can use the extractor provided by the assumption to efficiently extract $\boldsymbol{\mu}$ for every valid PoRet commitment.

The proof for the second part is given below. We provide two arguments based on whether a high-resolution / low-resolution anchor is used, beginning with the former setting.

As noted before, we assume that the clock offsets of all anchors are observed apriori and that clock drift is negligible. So we can safely assume that the anchor timestamps lie inside the expected interval, as otherwise the geo-commit verification would detect.

High-resolution anchors ($\boldsymbol{a = 1}$) Fixing some notation, assume that the storage provider P is at a location $L_p \in R^{\mathsf{target}}$ and that the anchor assigned to L_p is A, located at L_1. Recall that the target region in GoAT is a spherical circle centered at L_1 with radius $\delta = \Delta(L_p, L_1) \cdot S_{\mathsf{max}}/2$, i.e., the region $R^{\mathsf{in}} = (L_1; \delta)$. Expanding the radius further we have, $\delta = (t_{\mathsf{com}} + \mathsf{rtt}_{\mathsf{max}}(L_p, L_1) + t_{\mathsf{proc}}) \cdot (S_{\mathsf{max}}/2)$.

Recall that in the case of high-resolution anchors, the prover computes one PoRet commitment per interval. We want to prove that all the I PoRet commitments are computed on some device in R^{in}. Assume the contrary, i.e., say there exists a device $\mathsf{D}_{\mathsf{out}}$ situated at $L_2 \in R^{\mathsf{out}}$ on which one of the PoRet commitments is computed. By definition we have $\mathsf{dist}(L_1, L_2) > \delta$.

Without loss of generality, assume that a copy of the encoded file F^* (generated during the setup experiment) exists in its entirety in the memory of $\mathsf{D}_{\mathsf{out}}$, and therefore the time taken to compute commitment on $\mathsf{D}_{\mathsf{out}}$ is negligible, i.e., $t^{\mathcal{A}}_{\mathsf{com}} = 0$. We also set the anchor processing time $t^{\mathcal{A}}_{\mathsf{proc}} = 0$.

The time taken to receive and respond from $\mathsf{D}_{\mathsf{out}}$ during the geo-commitment protocol with A is given by $z = 2\mathsf{dist}(L_1, L_2)/S_{\mathsf{max}}$. This is because in Fig. 7 we derive challenges from anchor signatures, i.e., they arise at L_1 and must reach L_2. We can assume that the adversary probability of guessing these challenges is negligible (requires breaking selective unforgeability of the signature scheme used by the anchor which happens with negligible probability).

Note in particular that this value is irrespective of any other factors, e.g., the adversary's strategy might be to place a device D_{in} exactly at the anchor location L_1, and initiate the protocol from D_{in} with challenges forwarded to $\mathsf{D}_{\mathsf{out}}$. Moreover, we do not include any startup cost when the adversary is sending messages between devices, so $t^{\mathcal{A}}_{\mathsf{setup}} = 0$.

For the geo-commitment verification to succeed, it must be that $z \leq \Delta(L_p, L_1)$. (See last step in Fig. 2 when $a = 1$.)

But we have a contradiction, as z must also satisfy $z > 2\delta/S_{\mathsf{max}}$ because $\mathsf{dist}(L_1, L_2) > \delta$. Substituting for δ we get $z > \Delta(L_p, L_1)$. Hence proved. □

Low-resolution anchors ($a > 1$) The target region now has a slightly larger radius, $\delta = (\Gamma/a) \cdot S_{\mathsf{max}}/2$. The proof is very similar to the previous case. The main difference now is that the verification algorithm checks if $a+1$ anchor transcripts have the same time. Therefore the prover is forced to execute a PoRet commitments sequentially.

Recall that for low-resolution anchors, the prover computes a commitments every interval. Continuing in the same style as before, assume for contradiction that the prover tries to execute all the commitments in one of the intervals from $\mathsf{D}_{\mathsf{out}}$ ($\mathsf{D}_{\mathsf{out}}$ is setup in the same fashion as before).

The time difference between last and first timestamp in GeoCommit is given by $z = 2a\mathsf{dist}(L_1, L_2)/S_{\mathsf{max}}$. Note that we are counting time from the moment anchor receives the first request to the moment anchor sends out the last response.

To succeed in verification, it must be that $z \leq \Gamma$. Intuitively, this corresponds to $a+1$ timestamps having the same time. But we have a contradiction, as z must also satisfy $z > 2a\delta/S_{\mathsf{max}}$, substituting for δ we get $z > \Gamma$. Hence proved. □

Note on technique: One subtlety to note is that the following alternate amplification method that computes PoRet commitment only once does not work:

$ping_1$, com_1, $ping_2$, $ping_3$, ..., $ping_a$, $ping_{a+1}$. At first sight it might seem like a reasonable approach as it can also fill up a large amount of time.

But the proof does not go through because the adversary can decrease the time difference z as follows. Place $\mathsf{D_{in}}$ negligibly close to A and initiate the protocol from it. Therefore, the time taken for all consecutive pings is negligible. In this case, the timestamp difference will only be $z = 2\mathsf{dist}(L_1, L_2)/S_{\mathsf{max}}$ (incurred as the adversary would have to forward challenges required to compute com_1 from $\mathsf{D_{in}}$ to $\mathsf{D_{out}}$).

D.2 Remaining proofs

We now prove the remaining parts, part-one and part-three.

Part-one proof For this we reuse the proof for Theorem 4.2 in [38]. They provide a series of games that prove that, except with negligible probability, no adversary ever causes a verifier to accept in a PoRet instance, except by responding with values $\{\mu_j\}$,σ that are computed correctly (under the assumption that the computational Diffie-Hellman problem is hard in bilinear groups). This directly proves that if the challenger provides a challenge set $\mathcal{S}^*$, then the correctly computed output of SW.Prove and SW.Commit containing $\{C_{\boldsymbol{\mu}}, \boldsymbol{\mu}, \sigma\}$ must be accepted by the verification algorithm SW.Verify. The only change we made is the extra vector commitment. Assuming that the binding property of the vector commitment scheme holds, this directly follows.

The remaining thing to be proved is that all the individual PoRet commitments used to compute $C = C_{\boldsymbol{\mu}}$ are correctly computed. Assume for contradiction that some of them are not computed correctly. Observe that we derive random coefficients r_j from the final PoRet commitment com_N. These coefficients are used during verification to compute C as follows, $C = \prod_{j=1}^{N} (\mathsf{com}_j)^{r_N}$. Under the random oracle model, we can assume that the probability of prover guessing these coefficients beforehand is negligible. Note the two checks in SW.Verify: the commitment check (VC.Verify) and the pairing equation check. Assuming that the latter succeeds, that is the final commitment C is the same as that computed by an honest prover, then the only way prover can make VC.Verify succeed is by guessing the random coefficients correctly (or) by breaking commitment binding, both of which happen with negligible probability. Grinding concerns are discussed in the main body.

Part-three proof We re-purpose the extraction algorithm provided in the proof of Theorem 4.3 in [38]. [38] provides an extraction algorithm that, given an adversary that answers ϵ fraction of the queries correctly, can extract ρ fraction of the encoded file blocks provided that $\epsilon - (\rho n)^l/(n-l+1)^l$ is positive and non-negligible.

Recall that our extraction algorithm Extract is composed of Extr.Derive and Extr.Assemble. And the extraction algorithm of [38] already follows this addi-

Scheme	GoAT-H	GoAT-P
Parameters	$n = \frac{1}{b\rho}\lvert F\rvert$	$n = \frac{1}{s\rho\lambda}\lvert F\rvert$
Storage overhead	$32n + (\frac{1}{\rho} - 1)\lvert F\rvert$	$n\lambda + (\frac{1}{\rho} - 1)\lvert F\rvert$
Proof size	$Ial(b + 32\log_2 n) +$ $I(a+1)\lvert T\rvert$	$(s+1)\lambda +$ $I(a+1)\lvert T\rvert$
PoRet.Commit time	$O(lb)$	$O(ls)$

Table 3: Storage overhead, proof sizes and commit times for GoAT-H and GoAT-P.

tional structure we impose. Querying the adversary corresponds to Extr.Derive and assembling the file from query responses corresponds to Extr.Assemble.

The only change now is that extraction must succeed in every interval, i.e., $\mathcal{O}_{\mathsf{dev}}.\mathsf{seenInROU}(\mu_i^{\mathsf{all}}) = 1 \ \forall i \in \{1, 2, \ldots, I\}$ must pass for all the intervals. And the key question is how the new bandwidth constraint ϕ and grinding attacks (discussed in Sec. 4) impact the above theorem.

Recall that the size of the encoded file is $|F^*|$. Of this, due to grinding, at least $g = (1-(1-2^{-\lambda})^{1/\alpha})^{1/l}$ fraction is only stored inside R^{in} and hence only that is available for extraction (α is the grinding cap). And further, upto ϕ bytes (the bandwidth cap) of the g-sized fraction can be downloaded, and is hence unavailable.

The idea in the proof of Theorem 4.3 of [38] is to query enough times and use linear algebraic techniques to recover file blocks from query responses. Queries are made randomly. Three types of queries are listed, and the fraction of type-1 queries (the useful ones that help recover file blocks) is $\epsilon - w$ where $w = (\rho n)^l/(n-l+1)^l$ (omitting the negligible part of the equation caused by type-2 queries). The extractor needs $\rho n \leq n$ type-1 queries to succeed, which happens in $O(n/(\epsilon - w))$ time.

The maximum number of blocks unavailable inside R^{in} is given by $\gamma = (\frac{n\phi}{|F^*|}) + n(1-g)$. Therefore the extractor needs more type-1 queries to succeed, $(\rho n + \gamma)$. Note that we assume if a query challenges a block that belongs to the unavailable portion in S_1, a special symbol "-1" is used in place of the file block, and the challenge response is computed. And by extracting $(\rho n + \gamma)$ blocks, we are guaranteed to have at least ρn actual file blocks (removing the -1's).

The useful fraction of queries now is $\epsilon - w$ where $w = (\rho n + \gamma)^l/(n-l+1)^l$. And assuming $\rho n + \gamma \leq n$, extraction happens in $O(n/(\epsilon - w))$ time, i.e., same order as before. One constraint we get is $\frac{\phi}{|F^*|} \leq g - \rho$.

We want $\epsilon - w$ to be positive and non-negligible. Therefore w needs to be negligibly small. Meaning $(\rho + \gamma/n)^l$ (or) $(\rho + \frac{\phi}{|F^*|} + 1 - g)^l$ needs to be negligible. As noted above, the number of interactions required and the time to extract is the same order as in [38]. □

Proof of Geo-Retrievability

- $(sk,pk) \leftarrow \mathsf{KGen}(1^\lambda)$: Generate key pair. Run by the user.
- $(F^*, \eta, pp) \leftarrow \mathsf{Setup}(sk, pk, F)$: Runs setup of the underlying PoRet scheme to generate F^*, which contains the file plus the generated data, its handle η, and some public parameters pp. Run by the user.
- $c \leftarrow \mathsf{Chal}(\eta,pp)$: On input file handle η and params pp, derive a random challenge c. Run by the verifier.
- $\pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta,R,c)$: On input file handle η, a geographic region R and a challenge c, generate a proof of geo-retrievability π^{geo}. An interactive protocol between the prover and verifier.
  - $\mathsf{ComFrag}(\boldsymbol{\mu})$: Sub-function that operates on a file fragment $\boldsymbol{\mu}$.
- $0/1 \leftarrow \mathsf{Verify}(pp,R,c,\pi^{\mathsf{geo}},timer)$: The verifier checks that the file is in the desired region R by (a) verifying the proof π^{geo} using the challenge and the public params, and (b) checking that the measured time *timer* is bounded.
- $F \leftarrow \mathsf{Extract}(\eta,R,pp)$: The extraction algorithm consists of two sub-functions:
  - $\mu^{\mathsf{all}} \leftarrow \mathsf{Extr.Derive}(\eta, R, pp)$: On input file handle, geographic region, public parameters, run an interactive protocol with the prover P to output a list of file fragments μ^{all}.
  - $F \leftarrow \mathsf{Extr.Assemble}(\mu^{\mathsf{all}})$: Assemble file fragments to compute the file.

Fig. 5: Proof of Geo-Retrievability (PoGeoRet) API.

Non-interactive Proof of Geo-Retrievability

- $(sk,pk) \leftarrow \mathsf{KGen}(1^\lambda)$: Generate key pair. Run by the user.
- $(F^*, \eta, pp) \leftarrow \mathsf{Setup}(sk, pk, F)$: Runs setup of the underlying PoRet scheme to generate F^*, which contains the file plus the generated data, its handle η, and some public parameters pp. Run by the user.
- $\pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta,R)$: On input file handle η and a geographic region R, generates a proof of geo-retrievability π^{geo}. Run by the prover. It consists of two sub-functions:
  - $C^{\mathsf{geo}} \leftarrow \mathsf{GeoCommit}(\eta,R)$: On input file handle η and a region R, generate a geo-commitment C^{geo}. An interactive protocol between the prover and anchor. Furthermore, the protocol $\mathsf{com} \leftarrow \mathsf{ComFrag}(\boldsymbol{\mu})$ is a sub-function of $\mathsf{GeoCommit}$ which takes a file fragment $\boldsymbol{\mu}$ as input and generates a commitment com.
  - $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{PoRCompute}(\eta, \mathcal{S})$: On input file handle η and a set of PoRet challenges $\mathcal{S}$, compute one or more proofs of retrievability.
- $0/1 \leftarrow \mathsf{Verify}(pp,R,c,\pi^{\mathsf{geo}})$: The verifier checks that the file is in the desired region R by verifying the proof π^{geo} using the challenge, public params.

Fig. 6: NIPoGeoRet API.

E Supporting TLS 1.2 anchors

E.1 Low-resolution anchors

This section deals more broadly with supporting low-resolution anchors.

Chaining of the two operations is done in a similar fashion to before. In total, a PoRet commitment computations and $a+1$ anchor pings take place. We refer to a as the *amplification factor*. Note that this modification applies to both GoAT variants, GoAT-H and GoAT-P.

Geo-commitment generation (GeoCommit) between P and A

$\mathsf{Prot}_{\mathsf{A}}$: Follow the standard protocol (TLS 1.2 / Roughtime).

$\mathsf{Prot}_{\mathsf{P}}$: On input $\{\eta, pp\}$, runs the below protocol.

If $\Gamma_{\mathsf{A}} \leq 1$ms, set the amplification factor $a = 1$. Or else $a = \lfloor \Gamma_{\mathsf{A}} / \Delta(\mathsf{A},\mathsf{P}) \rfloor - 1$. Select a random N_1, $i = 1$ and do the following:

1. (**Anchor ping**) Request time from the anchor, $\{t_i, N_i, \sigma_i\} \leftarrow \mathsf{A}.\mathsf{GetAuthTime}(\mathsf{N_i})$. If $i = a+1$, then break.
2. (**PoRet commitment**) Run $\mathcal{S}_i \leftarrow \mathsf{PoRet.Chal}(\eta, pp, \sigma_i)$, $\mathsf{com}_i \leftarrow \mathsf{PoRet.Commit}(\eta, \mathcal{S}_i)$. Set $N_{i+1} = \mathsf{com}_i$, $i = i+1$ and repeat from step 1.

P saves $C^{\mathsf{geo}} = \{T_i\}_{i=1}^{a+1}$ where $T_i = \{t_i, N_i, \sigma_i\}$ denotes the transcript.

Geo-commitment verification by V

On receiving seed, epoch number e, interval number m, anchor public key pk_{A} and the geo-commitment $C^{\mathsf{geo}} = \left\{\{T_i\}_{i=1}^{a+1}, \{\mathsf{com}_i\}_{i=1}^{a}\right\}$, the auditor V does:

- Set $N_1 = \mathsf{seed}$ and $\forall i \in [1,\ldots,a], N_{i+1} = \mathsf{com}_i$.
- Verify anchor signatures using pk_{A}, $\forall i \in [1,\ldots,a+1]$, $\mathsf{Verify}_{pk_{\mathsf{A}}}(\sigma_i, \{t_i, N_i\})$ where $T_i = \{t_i, \sigma_i\}$.
- Check that the time corresponds to epoch e, interval m.
- Check that the timestamps are close:
  - If $a > 1$, check that the time is same, $t_1 = t_2 = \ldots = t_{a+1}$.
  - If $a = 1$, check that $t_2 - t_1 \leq \Delta(\mathsf{A},\mathsf{P})$.

Fig. 7: Geo-commitment protocols.

The value a is set based on the exact resolution offered by an anchor. For example if the anchor resolution is in seconds and the time difference $\Delta(L_{\mathsf{A}}, L_{\mathsf{P}})$ is 50ms, then 20 consecutive proofs (when started at a one-second boundary in the anchor's clock) will have the same timestamp, so $a = 19$ (since $a+1$ pings are needed). More generally, if the resolution of an anchor is Γ_{A}, we set $a = \lfloor \Gamma_{\mathsf{A}} / \Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \rfloor - 1$.[8] Below, we explain how to time proof execution in order to ensure receipt of $a+1$ transcripts with the same timestamp.

In Prove, the prover computes a single PoRet similar to before, leveraging the aggregability of SW. We also make a change to Verify: instead of checking the difference between timestamps, the verifier counts if $a+1$ anchor transcripts have the same timestamp. Other steps are similar to before.

A general GeoCommit protocol for *any* anchor, low- or high-resolution, is specified in Fig. 7, in the paper appendix.

When to start execution?: We have a question of when to initiate the protocol so that $a+1$ anchor transcripts have the same time. A simple approach is to continue executing proofs for roughly double the amplification factor a, specifically to use an augmented amplification factor $a' = 2\lceil \Gamma_{\mathsf{A}} / \Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \rceil - 1$. Irrespective

[8] In theory, $a = \lfloor \Gamma_{\mathsf{A}} / \Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \rfloor$ also works as $a \cdot \Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \leq \Gamma_{\mathsf{A}}$. But for perfect divisors, e.g., $\Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) = 50$ms, this can only be achieved with perfect time synchronization and ideal network conditions, making it impossible in practice.

Shacham-Waters PoRet scheme

Scheme parameters: A bilinear group $(p,\mathbb{G},\mathbb{G}^T,e,g)$. Number of challenges l. Number of sectors per block s. An erasure code with rate ρ. SW-H uses $\mathsf{VC}=\mathsf{HVC}$ and SW-P uses $\mathsf{VC}=\mathsf{CPVC}$.

- $(sk,pk) \leftarrow \mathsf{SW.KGen}(1^\lambda)$: Pick key pair $(\mathsf{ssk},\mathsf{spk}) \leftarrow \mathsf{KGen}(1^\lambda)$. Choose $\alpha \in \mathbb{Z}_p$ at random and compute $g^\alpha \in \mathbb{G}$. The secret key is $sk=(\mathsf{ssk},\alpha)$ while the public key is $pk=(\mathsf{spk},g^\alpha)$.
- $(F^*,\eta,pp) \leftarrow \mathsf{SW.Setup}(sk,pk,F)$: Apply erasure code over F to obtain F'. Split F' into n blocks, each s sectors long $\{m_{ij}\}_{1\le i\le n,1\le j\le s}$ with $m_{ij}\in\mathbb{Z}_p$. Pick s elements at random $\{u_i\}_{i=1}^s \in \mathbb{G}$. For each $i\in\{1,\dots,n\}$ compute $\sigma_i \leftarrow (H(i).\prod_{j=1}^s u_j^{m_{ij}})^\alpha$ where H is a hash-to-group function. Denote F^* as the file together with $\sigma_i, 1\le i\le n$. The public params[a] pp contains $\{pk,n,\{u_i\}_{i=1}^s\}$ along with a signature generated with ssk. $\eta=\mathsf{H}(F^*)$.
- $\{c_i,v_i\}_{i=1}^l \leftarrow \mathsf{SW.Chal}(\eta,pp,\mathsf{seed})$: Derive k values $c_i\in[n]$, $v_i\in\mathbb{Z}_p$ from the input seed. Return $\{c_i,v_i\}_{i=1}^k$.
- $C_{\boldsymbol{\mu}} \leftarrow \mathsf{SW.Commit}(\eta,\{c_i,v_i\}_{i=1}^k)$: Compute $\forall j\in[1,\dots,s], \mu_j \leftarrow \Sigma_{i=1}^k v_i m'_{ij}$ where $m'_{ij}=m_{(c_i)j}$. Commit to the vector $\boldsymbol{\mu}=\{\mu_j\}_{j=1}^s$ by $C_{\boldsymbol{\mu}} \leftarrow \mathsf{VC.Commit}(\boldsymbol{\mu})$.
- $\pi \leftarrow \mathsf{SW.Prove}(\eta,\{c_i,v_i\}_{i=1}^k)$: Compute $\sigma=\prod_{i=1}^k \sigma_{(c_i)}^{v_i}$ and $\forall j\in[1,\dots,s], \mu_j \leftarrow \Sigma_{i=1}^k v_i m'_{ij}$ where $m'_{ij}=m_{(c_i)j}$. Output $\pi=\{\boldsymbol{\mu},\sigma\}$ where $\boldsymbol{\mu}=\{\mu_j\}_{j=1}^s$.
- $0/1 \leftarrow \mathsf{SW.Verify}(pp,\{c_i,v_i\}_{i=1}^k,C_{\boldsymbol{\mu}},\pi)$: Receive $\pi=\{\boldsymbol{\mu},\sigma\}$. Check signature on t with spk and parse it receive $\{u_i\}_{i=1}^s$. Check if $e(\sigma,g)=e(\prod_{i=1}^k(H(c_i))^{v_i}\prod_{i=1}^s u_i^{\mu_i},pk)$. Check if $\mathsf{VC.Verify}(\boldsymbol{\mu},C_{\boldsymbol{\mu}})=1$.

[a] Referred to in the original Shacham-Waters paper as tag.

Fig. 8: The Shacham-Waters PoRet schemes with an extra commitment step. SW-H, SW-P differ in the choice of VC scheme.

of the start time in this case, the resulting sequence of transcripts are guaranteed to contain a $(a+1)$-length sub-sequence with the same timestamp (given stable network conditions). The final proof will only include the desired sub-sequence; extra transcripts can be discarded. The intuition here is that a' executions guarantees seeing two time changes (i.e., three distinct timestamps), therefore one resolution tick is fully covered, which in turn guarantees $\lfloor \Gamma_{\mathsf{A}}/\Delta(L_{\mathsf{A}},L_{\mathsf{P}})\rfloor$ transcripts will have the same timestamp. As noted, this will not work if the network conditions are unstable, and other mechanisms like retries are needed in practice.

Effect on geolocation: The use of amplification has a small effect on the radius of ROU, explained through an example. Suppose $\Delta(L_{\mathsf{A}},L_{\mathsf{P}})=250$ms, $\Gamma_{\mathsf{A}}=1000$ms. Applying the above formula, we get $a=3$, i.e., 4 pings are needed. But this leaves some "extra time"—for example, if the anchor's clock times at the moment of receipt of the 4 $\mathsf{GetAuthTime}$ requests are x, $x+250$, $x+500$, $x+750$ (all in ms), then an adversary still has about 250ms left in the end (Assume x is a second boundary). So an adversary can spend an extra $250/a=83.33$ms on each of the a PoRet commitment computations and thus position the file further from the target location than with no amplification. Such manipulation will go

Proof of Retrievability

- $(sk,pk)\leftarrow$ PoRet.KGen(1^λ): Generate key pair.
- $(F^*,\eta,pp)\leftarrow$ PoRet.Setup(sk,pk,F): F^* contains the encoded file, η denotes a unique file handle and pp contains the public parameters. We model the public key pk as a part of pp.
- $F\leftarrow$ PoRet.Extract(η,pp): An interactive function between a prover and verifier to recover original file F.
- $c\leftarrow$ PoRet.Chal$(\eta,pp,$seed$)$: Derive a challenge c from the input seed for the file η.
- $C\leftarrow$ PoRet.Commit(η,c): Generate a commitment C to the proof based on the challenge c.
- $\pi\leftarrow$ PoRet.Prove(η,c): Generate a proof π based on the challenge c.
- $0/1\leftarrow$ PoRet.Verify(pp,c,C,π): Verify both the commitment C and proof π using the public parameters pp.

Fig. 9: Publicly verifiable PoRet API. PoRet.Commit is the only addition compared to prior modeling [29].

Correlated Pedersen commitment CPVC

Params: Group $\mathbb{G}$ and it's support $\mathbb{Z}_p$. Supported vector size s. Generators $(h_1,h_2,...,h_s)\leftarrow\mathbb{G}$ and $(h_1^b,h_2^b,...,h_s^b)\leftarrow\mathbb{G}$.

- $C_{\mathbf{v}}\leftarrow$ CPVC.Commit$(\mathbf{v})$: Receive $\mathbf{v}=\{v_i\}_{i=1}^s$ where $\forall i, v_i\in\mathbb{Z}_p$. Output $C_{\mathbf{v}}=(\prod_{i=1}^s h_i^{v_i},\prod_{i=1}^s h_i^{bv_i})$.
- $0/1\leftarrow$ CPVC.Verify$(\mathbf{v},C_{\mathbf{v}})$: Check if $C_{\mathbf{v}}=(\prod_{i=1}^s h_i^{v_i},\prod_{i=1}^s h_i^{bv_i})$.

Hash-based commitment HVC

- $C_{\mathbf{v}}\leftarrow$ HVC.Commit$(\mathbf{v})$: Output $C_{\mathbf{v}}=\mathsf{H}(\mathbf{v})$.
- $0/1\leftarrow$ HVC.Verify$(\mathbf{v},C_{\mathbf{v}})$: Check if $C_{\mathbf{v}}=\mathsf{H}(\mathbf{v})$.

Fig. 10: Pedersen and Hash-based Vector Commitment scheme

undetected because the difference between the last and first anchor clock times is still within a resolution tick, $750+83.33\cdot 3=999.99\text{ms}<\Gamma_{\mathsf{A}}$.

The precise extra time available due to amplification is $e=\Gamma_{\mathsf{A}}-a\cdot\Delta(L_{\mathsf{A}},L_{\mathsf{P}})$. Distributing it equally leads to an extra e/a time per commitment computation. For practical values, the extra time is small and hence its impact is minimal. For example, if $\Delta(L_{\mathsf{A}},L_{\mathsf{P}})=50\text{ms}$ and $\Gamma_{\mathsf{A}}=1000\text{ms}$, then $e=50/19=2.6\text{ms}$ causing about 260km increase compared to that without amplification.

GoAT security: Considering both high-resolution and low-resolution anchors, the following equation describes GoAT's geolocation radii. Say the target region is a single location, $R^{\mathsf{target}}=(L;0)$. Then the region of uncertainty achieved by GoAT (both GoAT-H and GoAT-P) is a circle centered at anchor's location with radius δ_L given by:

$$\delta_L=\begin{cases}\Delta(L_{\mathsf{A}},L)\cdot S_{\mathsf{max}}/2 & \text{if } \Gamma_{\mathsf{A}}\le 1\text{ms}.\\ (\Gamma_{\mathsf{A}}/(\lfloor\Gamma_{\mathsf{A}}/\Delta(L_{\mathsf{A}},L)\rfloor-1))\cdot S_{\mathsf{max}}/2 & \text{otherwise.}\end{cases}$$

NIPoGeoRet scheme between U, P, V

Scheme parameters: List of anchors $\mathcal{T}$ and their corresponding public keys. Interval length β, number of intervals I.

$\mathsf{Prot}_{\mathsf{U}}$,$\mathsf{Prot}_{\mathsf{P}}$,$\mathsf{Prot}_{\mathsf{V}}$:

- $(sk,pk) \leftarrow \mathsf{KGen}(1^{\lambda})$: U runs $(sk,pk) \leftarrow \mathsf{PoRet.KGen}(1^{\lambda})$.
- $(F^*,\eta,pp) \leftarrow \mathsf{Setup}(sk,pk,F)$: U runs $(F^*,\eta,pp) \leftarrow \mathsf{PoRet.Setup}(sk,pk,F)$ and picks a geographic region $R=(L;\delta_L)$. Values $\{F^*,pp,R\}$ are given to P situated at L.
- $C^{\mathsf{geo}} \leftarrow \mathsf{GeoCommit}(\eta,R)$: P selects an anchor $\mathsf{A} \in \mathcal{T}$ based on the input region R and generates geo-commitments via GeoCommit (See Fig. 7) once every interval until the end of epoch. Note that $\mathsf{ComFrag}(\boldsymbol{\mu}) = \text{PoRet}.\mathsf{VC.Commit}(\boldsymbol{\mu})$.
- $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{PoRCompute}(\eta,\mathcal{S})$: Let $\mathcal{S} = \{\mathcal{S}_j\}_{j=1}^{N}$ denote the PoRet challenge sets derived in GeoCommit. The process differs by the PoRet scheme:
  - **SW-H**: Run $\forall j, \pi_j^{\mathsf{PoRet}} \leftarrow \mathsf{SW\text{-}H.Prove}(\eta,\mathcal{S}_j)$. $\pi^{\mathsf{PoRet}} = \{\pi_j^{\mathsf{PoRet}}\}_{j=1}^{N}$.
  - **SW-P**: P derives N random coefficients in $\mathbb{Z}_p$ from the last PoRet commitment, $\{r_j\}_{j=1}^{N} \leftarrow \mathsf{PRF}(\mathsf{com}_N)$. Denote $\mathcal{S}_j = \{c_{ij}, v_{ij}\}_{i=1}^{k}$. Apply random coefficients, $\forall j, \mathcal{S}_j^* = \{c_{ij}, r_j v_{ij}\}_{i=1}^{k}$ and merge all the sets to create, $\mathcal{S}^* = \cup_{j=1}^{N} \mathcal{S}_j^*$. Compute $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{SW\text{-}P.Prove}(\eta,\mathcal{S}^*)$.
- $\pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta,R)$: The two sub-functions of Prove are specified above. The proof of geo-retrievability π^{geo} consists of geo-commitments output by GeoCommit and the proof(s) of retrievability output by PoRCompute.
- $0/1 \leftarrow \mathsf{Verify}(pp,R,\pi^{\mathsf{geo}})$: Unpack $\pi^{\mathsf{geo}} = \{\{C_m^{\mathsf{geo}}\}_{m=1}^{I}, \pi^{\mathsf{PoRet}}\}$. The geo-commitments are verified first using the protocol in Fig. 7. Then the proof of retrievability π^{PoRet} is verified using the steps described below.
  - **SW-H**: Check $\forall j, \mathsf{SW\text{-}H.Verify}(pp,c_j,\mathsf{com}_j,\pi_j^{\mathsf{PoRet}})=1$ where $\pi^{\mathsf{PoRet}} = \{\pi_j^{\mathsf{PoRet}}\}_{j=1}^{N}$.
  - **SW-P**: Denote $\pi^{\mathsf{PoRet}} = \{\boldsymbol{\mu},\sigma\}$. V generates random coefficients $\{r_j\}_{j=1}^{N}$ and aggregate challenge set $\mathcal{S}^*$ similar to how P does in Prove. V computes $C = \prod_{j=1}^{N} (\mathsf{com}_j)^{r_j}$ and checks if $\mathsf{SW\text{-}P.Verify}(pp,\mathcal{S}^*,C,\pi^{\mathsf{PoRet}})=1$.

Fig. 11: The GoAT proof of geo-retrievability schemes. It includes both the GoAT-H and GoAT-P variants that internally use SW-H and SW-P PoRet schemes respectively.

Merkle tree PoRet scheme

Scheme parameters: Block size b and number of challenges l. An erasure code with rate ρ.

- $(sk,pk) \leftarrow \mathsf{MT.KGen}(1^{\lambda})$: Run $\mathsf{KGen}(1^{\lambda})$.
- $(F^*,\eta,pp) \leftarrow \mathsf{MT.Setup}(sk,pk,F)$: Apply erasure code over F to obtain F'. Split F' into n blocks ($n = |F'|/b$) and build a Merkle tree. Denote the tree root by r. Set $pp = \{n,r,pk,\sigma_r\}$ where $\sigma_r = \mathsf{Sig}_{sk}(n\|r)$. Set F^* to F' plus the Merkle tree and $\eta = \mathsf{H}(F^*)$.
- $\{c_i\}_{i=1}^{k} \leftarrow \mathsf{MT.Chal}(\eta,pp,\mathsf{seed})$: Derive k values from the input seed in the range $[1,n]$.
- $C \leftarrow \mathsf{MT.Commit}(\eta,\{c_i\}_{i=1}^{k})$: For each challenge c_i, retrieve the file block f_i using the file handle η and compute $C = \mathsf{H}(\{f_i\}_{i=1}^{k})$.
- $\pi \leftarrow \mathsf{MT.Prove}(\eta,\{c_i\}_{i=1}^{k})$: For each challenge c_i, retrieve the file block f_i and its sibling path p_i in the Merkle tree. Set $\pi = \{f_i,p_i\}_{i=1}^{k}$.
- $0/1 \leftarrow \mathsf{MT.Verify}(pp, \{c_i\}_{i=1}^{k}, C, \pi)$: Expand $pp = \{n,r,pk,\sigma_r\}$. Check $\mathsf{Verify}_{pk}(r,\sigma_r)=1$. $\forall 1 \le i \le k$, check each triple $\{c_i,f_i,p_i\}$ using the root r.

Fig. 12: The Merkle tree PoRet scheme with an extra commitment step.

E.2 Changes to ComFrag

One other change needs to be made to support TLS. In GoAT-P, the vector commitment has two elements and won't fit into the nonce field of the TLS handshake for commonly used algebraic groups. So we include a hash of the commitment and reveal the underlying commitment as part of the proof. Details below.

For example, the size of each group element in our implementation is 20 bytes, so the SW-P PoRet commitment is 40 bytes whereas the TLS nonce is 32 bytes only.

So we modify the PoRet commitment protocol by hashing the previous commitment to fit in the nonce field (which is essentially in turn modifying the ComFrag protocol). The output of the Prove protocol, i.e., the PoGeoRet proof will now include all the PoRet commitments generated during the epoch.

If the number of intervals is 1, the proof will consist of a proof-of-retrievability, a PoRet commitments and $a+1$ anchor transcripts.

F Formalism Extensions

F.1 Non-interactive Proofs of Geo-Retrievability

NIPoGeoRet allows any newcomer to verify that the prover indeed had the file inside the region of uncertainty (ROU), during a specified time duration. The NIPoGeoRet API (Fig. 6) is almost the same as the PoGeoRet one except that the function Chal is removed. We attach the preamble NI to other API functions, e.g., Prove and Verify.

Relation to GoAT: Recall that GoAT is a non-interactive protocol. So the API in Fig. 6 map to the GoAT protocol specified in Fig. 11.

For ease of explaining GoAT, we divide Prove into two sub-functions, GeoCommit and PoRCompute. The former specifies the interaction with anchor A to derive challenges. GeoCommit for GoAT is specified in Fig. 7.

Modeling time: In our previous modeling for interactive PoGeoRet, we relied on the verifier to keep track of time during the security experiments. Instead now we introduce a notion of time into the definition. Each system entity maintains an internal clock. Clocks need not be synchronous, but we assume that clock drift is negligible. The clock time of say an anchor A is given by $\mathsf{time}_{\mathsf{A}}$. If the true time is given by true_time, then the clock offset of an entity A is $(\mathsf{time}_{\mathsf{A}} - \mathsf{true\_time})$. The offsets of all anchors are assumed to be public (this can be observed once during a setup phase in practice). Note that the Verify function relies on these clock offsets to judge if the proof is valid.

Security properties: The completeness definition is the same as before, except that no challenges are issued by the verifier.

The changes to the security experiments related to soundness are also minimal. The setup experiment is same as before, except that the public information pp could also contain extra information such as anchor public keys. The challenge experiment now does not involve sending challenges to the prover. Instead,

File size (MiB)	Encoding time (ms)
32	100 (3.37)
64	334.6 (6.93)
128	756.8 (4.78)
256	1648.6 (27.26)
512	3803.2 (38.13)
1024	8738.7 (51.11)

Table 4: Reed Solomon encoding time with a symbol size of 32 bytes. Averaged over 10 runs.

the prover computes NIPoGeoRet proofs itself, and submits a proof at the end of an epoch. This proof is verified using Verify. And the soundness definition is the same as before.

G Miscellaneous

G.1 Practical considerations

Grinding attacks: Since GeoCommit protocol is prover-initiated, an adversarial prover can exploit by re-running the protocol. For example, an adversary could save on storage by only storing a portion of the file, and repeatedly query the anchor until all the challenges lie in the stored part.

Let g be the stored fraction. To model practical constraints, we assume that a prover can make upto 2^α GetAuthTime API calls per interval (this number needs to be set based on the actual API call costs). The success probability after 2^α API calls is $p = 1-(1-g^l)^{2^\alpha}$. The adversary needs to choose the file-fraction g such that p is non-negligible, i.e., $g \geq (1-(1-2^{-\lambda})^{2^{-\alpha}})^{1/l}$ (or) $g > 2^{\frac{-\lambda-\alpha}{l}}$ (via binomial expansion). Intuitively as the number of challenges l is raised, the adversary is forced to store more. We derive an exact constraint involving l and α in our security proofs.

Coefficient randomization: Randomization at the end of an epoch is necessary to ensure that the PoRet commitments $\{\mathsf{com}_i\}$ are correctly computed in all intervals. If the ratio between any two random coefficients was predictable, e.g., say $\tau = r_i/r_j$ was known for some $i<j$, then an adversary could cheat by postponing file access required to be done in the ith interval to the jth interval. Simply set com_i to random and com_j in a way that the verification equation checks out, i.e., $\mathsf{com}_j = (H_i(\mathsf{com}_i)^{-1})^\tau H_j$. H_i and H_j are the actual ith and jth PoRet commitments that the adversary computes in the jth interval. More formally, we later show that an adversary that skips PoRet commitments cannot succeed in verification, as it is equivalent to breaking commitment binding, which can happen with negligible probability.

We ensure a negligible likelihood of guessing the random coefficients $\{r_j\}$ a priori by deriving them from the final PoRet commitment com_I. This still leaves

Protocol	Anchor type	Proof size
GoAT-H	Any	$Iae(s+1)+I(a+1)\|T\|$
GoAT-P	TLS 1.2	$(s+1)e+2Iae+I(a+1)\|T\|$
GoAT-P	Roughtime	$(s+1)e+2I\|T\|$

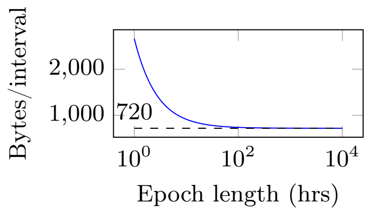

Fig. 13: (Left) GoAT proof sizes. e is the size of elements in $\mathbb{G}$ or $\mathbb{Z}_p$ of the Shacham-Waters PoRet scheme. (Right) Proof size per interval of GoAT-P with a Roughtime (RT) anchor against the epoch length. Interval length $\beta = 1$hr. Dashed line (720B) corresponds to the size of two RT transcripts.

possible grinding attacks. The best strategy for an adversary is to randomly choose the commitments (or random coefficients) and check if the verification equation succeeds. The probability of success is $2^{-\lambda}$ (as 2^λ is the size of the group used). With grinding, the probability increases to $2^{-\lambda+\alpha}$, which is still negligible for practical parameters. One way to avoid grinding is to obtain random coefficients from public randomness beacons, e.g., [21].

Parameterization trade-offs: We discuss various trade-offs arising in GoAT parameterization now.

The number of sectors s impacts the proof sizes, geolocation quality and the storage overhead. Higher s leads to reduced storage overhead but at the cost of relatively poorer geolocation and worse proof size. Note that higher s leads to increased PoRet commit times and thereby worse geolocation (eq. (1)).

The number of challenges k and the code rate ρ need to be set following the constraint given in Thm. 1. As shown in Sec. 5, for practical values of ρ, the number of challenges is around 200. k and ρ impact geolocation quality and storage overhead respectively. There is a direct trade-off between the two — higher code rate (ρ) leads to less storage overhead but requires setting a higher number of challenges (k), which leads to higher PoRet commit times and worse geolocation.

G.2 Implementation details

Anchor processing times: Many TLS servers take a non-negligible amount of time to compute the response, called the anchor processing time (t_{aproc}). This is measured by pinging 114 servers at repeated intervals over two weeks both via TLS (with TCP connections established apriori) and ICMP (for raw RTT). The processing time is defined as the difference between the two. We compute the average processing time for each server, and then the 75th percentile over all the servers, which is $t^{\mathsf{tls}}_{\mathsf{aproc}} = 6.5$ms. Anchors in the remaining 25th percentile are discarded. Note that setting a somewhat high value of 6.5ms for *all* TLS servers is conservative—a better approach is to set anchor-specific values.

For Roughtime, we find that the processing times are almost negligible, we set $t^{\mathsf{rt}}_{\mathsf{aproc}} = 2$ms. This could be due to a combination of several factors, e.g., less load, faster transport layer (UDP) [17] and faster signature scheme (EdDSA).

Remaining parameters: Two more parameters remain to be set: t_{proc} and t_{com} (see eq. (1)). t_{proc} is separated into client (t_{cproc}) and anchor (t_{aproc}) components, with the latter discussed before. t_{cproc} corresponds to the time spent in handling the anchor response. We set $t_{\mathsf{cproc}} = 1.5$ms and $t_{\mathsf{com}} = 2$ms based on code benchmarks (the latter is discussed below).

Erasure codes: Recall that a PoRet encoding F^* of file F incorporates an erasure code (to amplify soundness). Our implementation omits this part as it is standard to all PoRet implementations.

An (N,K)-erasure code encodes a message consisting of K symbols into another message of K symbols such that the original message can be recovered from a subset of the K symbols. The code rate ρ is equal to K/N. Note that a symbol here is same as that defined in the explanation of the Shacham-Waters scheme in Sec. 4. In our implementation, the symbol size is 32 bytes. For a fixed ρ and file size $|F|$, observe that we want an erasure code with $K = |F|/32$ and $N = K/\rho$.

RS encoding is expected to run in $O(n \log n)$ time. We use an off-the-shelf library that implements Reed-Solomon codes with large message, block lengths. Table 4 presents the execution results from running RS encoding on a c5.4xlarge AWS machine (like before). We set the symbol size to 32 bytes and modify the field appropriately to allow for larger N values. It takes about 8 seconds to encode a 1GB file, and more generally, a slightly superlinear growth can be observed in the timing numbers matching the expected shape of $O(n \log n)$.

Some directions of future work include exploring optimizations that can reduce encoding times for larger file sizes or exploring settings where the user can offload erasure coding to another entity.

G.3 Ways to improve geolocation accuracy

One set of ideas is related to improving the network model. Our current network model is unified, i.e., it assumes the network conditions across the globe are same for simplicity. Taking endpoint locations into account can improve geolocation quality in areas with better connectivity. Moreover a network model that avoids the blanket use of a startup cost t_{setup} (we set it to 5ms) is desirable given that it causes upto 2x worse geolocation for nearby anchors. In our small measurement study, we found a lot of variance in the round trip times for nearby locations. But since GoAT can deal with short-lived network variances better due to the use of flexible-challenge model, a smaller value for t_{setup} could be used. More experiments to understand if this idea can be used in practice are needed.

The network model could also be more nuanced, for example nearby locations are known to have higher latencies due to long routing paths. In this case, choosing a different model based on how close the two locations are would be better.

Another idea is to optimize the PoRet commit compute time (we set it to 2ms). This can for example be done by finding a pairing-friendly curve that has faster vector commit times and optimizing code runtime.

With regards to the choice of anchors, using Roughtime servers is clearly beneficial if possible due to their low processing times. Otherwise finding TLS servers that respond quickly is suggested, i.e., have low processing times. Overall Roughtime is a better choice of anchor, both from a performance perspective and an ethical standpoint since our use of TLS might be seen as abusing it. We hope that Roughtime gains more adoption in the future.

Several other optimization opportunities exist: reducing the client processing time by optimizing client-code (we allocate 1.5ms which could potentially be reduced to almost zero), using an anchor-specific model for processing times, and perhaps even deploying new anchors with fast connectivity and low processing times.

H GoAT-H: GoAT with Merkle trees

We now provide details of an alternate construction, GoAT-H, that uses Merkle Trees as the Proof of Retrievability scheme. We refer to the previously proposed GoAT as GoAT-P.

H.1 Merkle Tree PoRet

We begin with a brief explanation of the Merkle tree PoRet scheme and the newly added commit function.

The setup phase (MT.Setup) is as follows. The user divides the file F into blocks and builds a Merkle tree with the blocks serving as leafs of the tree. The root of the Merkle tree r is signed by the user, and the resultant tuple $pp{=}\{r,\sigma_r\}$ forms the public parameters.

A PoRet challenge consists of k file block indices. For each block index requested, the proof of retrievability consists of the requested block together with a sibling path to the root of the Merkle tree. Any party can later verify the proof later using just the public parameters pp.

The new commit function, MT.Commit, involves computing a hash over the requested file blocks; note that the Merkle tree is not used in this computation. Figure 12 presents the scheme.

H.2 GoAT-H protocol

Figure 11 also specifies GoAT-H, denoted by "MT" in the figure. The GeoCommit protocol is same as before. In Prove, unlike GoAT-P, all PoRets are computed individually thereby leading to high proof sizes. The proofs are later verified in Verify.

Table 3 presents the concrete storage overhead, proof size and the algorithmic complexity of commit for GoAT-H and GoAT-P. We note that the proof sizes shown in the table does not account for several practical optimizations one might use to compress GoAT-H proofs, e.g., remove some portion of the top half of the tree as it is likely the same. We expect these optimizations to bring down proof sizes by a small constant only.