GoAT: File Geolocation via Anchor Timestamping

ABSTRACT

Blockchain systems are rapidly gaining traction. Decentralized storage systems like Filecoin are a crucial component of this ecosystem that aim to provide robust file storage through a Proof of Replication (PoRep) or its variants. However, a PoRep actually offers limited robustness. Indeed if all the file replicas are stored on a single hard disk, a single catastrophic event is enough to lose the file.

We introduce a new primitive, *Proof of Geo-Retrievability* or in short PoGeoRet, that enables proving that a file is located within a strict geographic boundary. Using PoGeoRet, one can trivially construct a PoRep by proving that a file is in several distinct geographic regions. We define what it means for a PoGeoRet scheme to be complete and sound, in the process making important extensions to prior formalism.

We propose GoAT, a practical PoGeoRet scheme to prove file geolocation. Unlike previous geolocation systems that make strong assumptions about storage providers and require dedicated anchors, GoAT makes minimal assumptions and uses existing timestamping servers on the internet as geolocation anchors. 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 $2 trillion market at the time of writing [3]. Many decentralized applications, ranging from non-fungible tokens (NFT) [22] to retention of blockchain state [5], 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 [15], Filecoin [33], Sia [45], Storj [34], etc.

Like today's cloud storage services (e.g., Amazon S3 [11]), 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. Most notably, Filecoin [17] uses a protocol called Proof of Replication (PoRep) [24] for this purpose, while related systems such as Sia, Storj, etc., use similar techniques.[1]

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 a PoRep *does not 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.

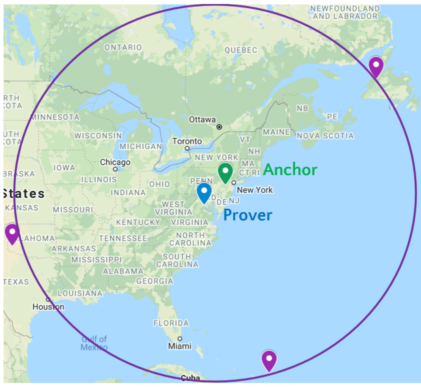

Figure 1: A prover and a Roughtime anchor are situated 300km apart. GoAT's region of uncertainty is a circle of radius 2000km (purple) proving that the replica is in the east half of North America.

In this paper, we explore an alternative approach to building PoReps: 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 proof automatically implies the property ensured by a PoRep, 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 PoReps, 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., [28].[2] It can also be used by CDN providers to prove that they are serving data from geographically distributed locations according to a claimed policy.

The goal of our work is, specifically, 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-mile diameter), which suffices for key applications such as file replication.

We introduce a formal definition of PoGeoRets in this paper, and propose, implement, and experimentally validate a PoGeoRet

[1] Filecoin has the most flexible yet most computationally expensive approach among these systems: Its PoRep proof system works for plaintext files, while other decentralized storage systems only work assuming distinct ciphertext file replicas.

[2] Some nations only require that a copy of data be stored locally whereas more stricter laws make transferring data abroad illegal [28]. Our techniques suffice for the former but the latter would additionally require the use of trusted hardware.

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

Previous works have explored internet-resource geolocation—both servers [32, 46] and files [18, 47]—but operate in benign cloud settings, e.g., assume cloud providers have normal network connectivity (even between two datacenters [18]), verifiers are close to providers, and/or files are stored in cloud systems whose locations are known *a priori*, etc. These assumptions make such approaches unsuitable for decentralized settings of the type we explore here. GoAT requires none of these assumptions.

1.1 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.

Anchors used in GoAT need not be in close proximity to storage service providers. Additionally, the main job of an anchor is *not* to geolocate entities directly, but only to provide timestamps. It is possible to handle a local minority of misbehaving anchors.

Anchors can be purpose-built for a GoAT instance. We also show, however, that it is possible to use *existing, unmodified* servers, e.g., TLS 1.2 or Roughtime [9] servers, as anchors. Thus it is possible to realize GoAT with *today's internet infrastructure*.

1.2 Proving Geolocation

In GoAT, a prover must prove proximity to an anchor. The starting point for GoAT is a simple, well known technique: The prover *pings the anchor successively* to get two timestamps t_1, t_2. If the prover is indeed situated close to the anchor, then the timestamps will not differ by much, i.e., $t_2 - t_1 < \Delta$ for some small Δ.[3] Identification of the prover is done by signing the first anchor response with the prover's private key and using the signature as a nonce in the second ping. This form of chaining is also crucial to ensure that the two anchor pings are indeed made successively.

The two anchor responses together form a proof that the prover is situated close to the anchor. The signature on these responses makes the proof publicly verifiable. Assuming that the anchor location is known, it becomes a proof of location for the prover.

GoAT requires that a majority of anchors situated near a given location be honest. Intuitively, this localization is necessary since each anchor is only useful in places close to its location.

Realizing proofs of geolocation: Several challenges arise in basic proofs of geolocation. Existing anchors pose one key challenge. 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 a technique for *amplification*: Instead of pinging twice, a prover pings the anchor repeatedly with interrelated challenges over an extended time interval, e.g., a full 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.

Another important practical concern is handling network volatility. We provide an empirical framework to calibrate the time threshold Δ for the prover to assert proximity to an anchor. The framework depends on factors like expected network quality and anchor characteristics, such as how quickly a given anchor responds. Our approach helps minimize false rejection rates, particularly given that in our protocols a brief period of good network connectivity (say few seconds) amidst a longer period of time (say hours) suffices for an honest prover to prove file replica possession successfully.

1.3 Geolocating Files: GoAT

To build on basic geolocation proofs and realize GoAT, our strategy is to interleave into the prover's anchor pings a Proof of Retrievability (PoRet) [31, 40]. A PoRet proves storage of a full file replica. In isolation, though, it proves nothing about a file's storage location. Thus the need to integrate it with a geolocation scheme.

Making GoAT efficient: A key challenge is reducing GoAT's communication complexity. Due to a combination of amplification, proof accumulation over several epochs and different anchors, the proof sizes quickly blow up, even with use of the communication-efficient Shacham-Waters (SW) PoRet [40]. Through incorporation of vector commitments and compression across proof instances, we manage to compress the size of PoRet-related proofs even across a sequence of pings to just a few bytes.

Yet another challenge in realizing GoAT is minimizing the time taken for the operation between the two anchor pings. Slow operation—as caused by computing a full PoRet proof—degrades geolocation accuracy. We therefore introduce techniques to compute a fast PoRet commitment to the randomness in a SW proof between the two pings, without actually computing the proof. As we will show later, the introduction of PoRet commitments is crucial for geolocation, improving accuracy by as much as 20x in some cases.

Figure 1 illustrates GoAT's file geolocation capabilities.

Defining PoGeoRet: A theoretical contribution of our work is in defining what it means for a PoGeoRet scheme to be secure. 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. 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. We say that a PoGeoRet is sound if the file fragments seen by the commitment oracle are enough to recompute the file.

The choice of PoRet commitment function leads to two variants of GoAT with a performance, security assumption tradeoff between them. GoAT-H uses a hash function to commit and admits a security proof under common assumptions. GoAT-P uses homomorphic

[3] This is not always true due to network abnormalities. GoAT accounts for them with a conservative network model introduced later.

commitments to achieve at least 3x smaller proof sizes but relies on a new knowledge assumption closely related to KEA1 [14].

Contributions: Our contributions are summarized as follows:

(1) *New Security Definitions and Modeling*: We define what it means for a PoGeoRet scheme to be complete and sound, the latter requiring important extensions to the classic PoRet security experiments (Sec. 3). We also introduce practical model variants for PoRet and PoGeoRets of potential independent interest that facilitate bootstrapping using existing servers and fast encoding.

(2) GoAT: We introduce our Proof of Geo-Retrievability (PoGeoRet) protocol GoAT in Sec. 4. GoAT leverages the Shacham-Waters PoRet and timestamping anchors. We explore optimizations to reduce the size of GoAT proofs, achieving constant-size PoRet-component in our proofs. We prove GoAT security in App. D.

(3) *Implementation and Evaluation*: To demonstrate GoAT's practicality we prototype 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. We show geolocation radii lower than 1000km, even tighter than required for applications like file replication (Sec. 5).

Sec. 6 contains related work. We have also released GoAT as an open-source tool.

2 PRELIMINARIES

2.1 Authenticated time protocols

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

2.1.1 TLS 1.2. Some TLS 1.2 servers [23] 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 trick works for Diffie-Hellman based key-exchange, including elliptic-curve variants, and for RSA as well.

This functionality has always been an informal practice, and is not specified in the TLS 1.2 RFC, but is widely practiced—we found about 1/5 of top 500 hosts in Alexa list 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 in the near future, e.g., in April 2021, 99.4% of Alexa top 1M sites [1] were found to support TLS 1.2 [38].

2.1.2 Roughtime. Roughtime [9] is a recently developed authenticated time protocol. At the time of writing, we are aware of four providers hosting Roughtime—Cloudflare [37], Google, Chainpoint and int08h. Roughtime servers provide a highly precise timestamp in 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.

2.2 Proof of Retrievability

Proof of Retrievability [31] schemes enable a prover to prove knowledge of a complete file replica in a communication-efficient manner. For GoAT, we require a publicly verifiable PoRet scheme. Merkle-tree (MT) based variants [31] and Shacham-Waters (SW) [40] are the two main choices.

Figure 10 shows the API for a PoRet scheme. The file owner begins by generating a key pair. The setup protocol takes an input file F and outputs a transformed file F^* which contains the file plus erasure-coding data and some extra data to support the proofs. The setup protocol also outputs a unique handle η for the file and some public parameters pp. In a typical PoRet system, pp is posted publicly so that any party can verify a proof of retrievability.

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. The PoRet.Commit function is the only addition we make to the PoRet scheme used in GoAT, which otherwise remains unmodified.

3 FORMALIZING PROOFS OF GEOGRAPHIC RETRIEVABILITY

A Proof-of-Geographic-Retrievability (PoGeoRet) scheme includes three parties[4]: a *user* (U) that 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 the storage claims of storage providers.

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 a correct location. Soundness means that any dishonest prover either not storing the complete file or storing it outside a permitted geographic boundary should be detected with high probability.

Section structure: We start with preliminaries in Sec. 3.1, explaining how a PoGeoRet leverages an underlying PoRet. We provide the adversarial model in Sec. 3.2, and then present the basic modeling behind our formal definitions. We formalize completeness in Sec. 3.3 and soundness in Sec. 3.4. Finally, in Sec. 3.5, we discuss modifications to our security model and definitions that we believe reflect requirements in real-world use cases, such as support for fast file encoding and easy bootstrapping.

3.1 Preliminaries

Protocol structure: The API for a PoGeoRet scheme is in Fig. 2.

We assume in this API and throughout this section that a PoGeoRet scheme internally leverages a PoRet scheme. In what follows, where clear from context, we drop PoGeoRet from our notation, e.g., use Setup to denote PoGeoRet.Setup.

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

[4] Of course, in practice, a decentralized system will typically include many instances of each party type.

Proof of Geo-Retrievability

- $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{KGen}(1^\lambda)$: Generate key pair. Run by the user.
- $(F^*,\eta,\mathsf{pp}) \leftarrow \mathsf{Setup}(\mathsf{sk},\mathsf{pk},F)$: 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.
- $\mathsf{c} \leftarrow \mathsf{Chal}(\eta,\mathsf{pp})$: On input file handle η and params pp, derive a random challenge c. Run by the verifier.
- $\pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta,R,\mathsf{c})$: On input file handle η, a geographic region R and a challenge c, generate a proof of geo-retrievability π^{geo}. Run by the prover.
  - $\mathsf{Commit}(\boldsymbol{\mu})$: A sub-function of Prove that operates on a file fragment $\boldsymbol{\mu}$.
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{pp},R,\mathsf{c},\pi^{\mathsf{geo}})$: The verifier checks that the file is in the desired region R by verifying the proof π^{geo} using the challenge, public params.
- $F \leftarrow \mathsf{Extract}(\eta,R,\mathsf{pp})$: The extraction algorithm consists of two sub-functions:
  - $\mu^{\mathsf{all}} \leftarrow \mathsf{Extr.Derive}(\eta,R,\mathsf{pp})$: An interactive protocol run with the prover. It takes as input file handle, geographic region, public parameters and outputs a list of file fragments μ^{all}.
  - $F \leftarrow \mathsf{Extr.Assemble}(\mu^{\mathsf{all}})$: Assemble file fragments to compute the file.

Figure 2: Proof of Geo-Retrievability (PoGeoRet) API.

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.

A PoGeoRet protocol runs in *epochs*. During each epoch, the provider P computes a *Proof of Geo-Retrievability* using the Prove protocol. An auditor V can use the public parameters pp to verify the generated proof via the Verify protocol.

The key ingredient in Prove enabling a file geolocation proof is the sub-function Commit. It takes a file fragment $\boldsymbol{\mu}$ as input and outputs a commitment of it.

A PoGeoRet must also specify an extraction algorithm Extract that can recompute the file F from the prover's responses. Extract will be used to model extraction in the soundness definition of a PoGeoRet in a way largely similar to prior works [31, 40]. A key difference from prior works is that Extract needs to follows a specific design; it must be composed of two algorithms: Extr.Derive interacts with the prover and outputs a list of file fragments μ^{all} and Extr.Assemble recomputes the file F from the fragments. We assume, as in, e.g., [31, 40], that during extraction, the prover can be *rewound*.

Modeling geolocation: We model geolocation in a PoGeoRet using a metric space [7] $(\mathcal{M},\mathsf{dist})$ where $\mathcal{M}$ is the full set of possible storage locations and dist is a distance metric[5] 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 a location $L \in \mathcal{M}$, we define a *region* $R=(L;\delta)$ as the set of all $L' \in \mathcal{M}$ that satisfy $\mathsf{dist}(L,L') \le \delta$. For example, when $\mathcal{M}$ models points on a sphere, regions correspond to circles on the surface. For simplicity, we will only consider such circular regions. We use the notation $R.L$ / $R.\delta$ to refer to the center / radius of R respectively.

Suppose that we want the PoGeoRet scheme to facilitate storage of files in a target region $R^{\mathsf{target}}=(L;\delta)$ where δ is a small radius that captures the breadth of the target region. Our definition allows for any arbitrary δ; in practical settings however, geolocation will be most beneficial for a small target region, e.g., the size of a city. Any storage provider located inside R^{target} can then join the system.

Region of uncertainty: We define a *Region Of Uncertainty* (ROU) denoted R^{rou} to capture the permitted noise in the attained geolocation guarantee. The PoGeoRet scheme then must ensure that files are stored inside the region R^{rou}. In other words, an ROU helps eliminate spurious proof failures.

Notation	Description
U	User / File owner
P	Storage provider / Prover
V	Auditor / Verifier
$\mathsf{A},\mathcal{T}$	Anchors (single / set)

Table 1: System entities. Anchors are specific to GoAT.

Continuing the previous example of earth surface as $\mathcal{M}$, say we want 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 $\delta=10\text{km}$.[6] Suppose that we are willing to tolerate noise in proofs up to the point where we ensure that files are at most 1000km from New York. The desired region of uncertainty then is $R^{\mathsf{rou}}=(L;1000\text{km})$.

Our definitions are given with respect to a single target region R^{target}. In practice, it might be desirable to support several distant locations. We expect our definition to be applied to each desired target region independently.

Where convenient, we refer to a region of uncertainty R^{rou} as R^{in} and define its complement by $R^{\mathsf{out}}=\mathcal{M}\setminus R^{\mathsf{in}}$.

Storage devices: To allow an adversary to place files in several distinct locations, we introduce a model of *(storage) devices*. We denote a device by D. In our security experiments (for soundness), all devices are under the control of the adversary / prover. The prover can place devices in locations of its choice but those locations remain fixed throughout the experiment. Devices have access to unlimited storage memory. Formally, we model all devices by way of an oracle $\mathcal{O}_{\mathsf{dev}}$ presented in Sec. 3.4.

Modeling time: As noted before, 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. To do so, each critical operation involved—both computation and communication—must have a specified expected time. We allow the adversary to communicate messages (of any size) between devices with speed S_{max}. In our security experiments below, we assume that the verifier internally keeps track of time whenever necessary.

3.2 Adversarial Model

Table 1 lists the entities in a PoGeoRet scheme. The adversary $\mathcal{A}$ controls the storage provider P. We assume that the auditor V and the user U are honest. (In a decentralized system, it is easy to imagine an honest-majority assumption for auditors.)

3.3 Completeness

Completeness requires that for all key pairs $(\mathsf{sk},\mathsf{pk})$ output by KGen, for all files $F \in \{0,1\}^*$, and for all $\{F^*,\eta,\mathsf{pp}\}$ output by $\mathsf{Setup}(\mathsf{sk},\mathsf{pk},F)$, the verification algorithm accepts when interacting with a valid prover P situated on a device D inside R^{target}, i.e., for all challenges $\mathsf{c} \leftarrow \mathsf{Chal}(\eta,\mathsf{pp})$, we have

$$\Pr\left[1 \leftarrow \mathsf{Verify}(\mathsf{pp},R^{\mathsf{in}},\mathsf{c},\pi^{\mathsf{geo}}) \middle| \pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta,R^{\mathsf{in}},\mathsf{c})\right] \ge 1-\mathsf{negl}(\lambda).$$

[5] The metric is a function that defines the concept of a distance between any two set members, and satisfying a few simple properties such as the triangle inequality.

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

State: A region $R^{\mathsf{in}}$. Key-value pairs $\mathcal{D}[\mathsf{did}] = (\mathsf{loc},\mathsf{mem})$ where the key did is the device identifier, loc is its location, mem is a list denoting the memory. A list $\mu^{\mathsf{rec}}$ to track inputs to the commitment oracle $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$.
$\mathsf{init}(R)$: Set $R^{\mathsf{in}} = R$. Not callable by the adversary.
$\mathsf{createDevice}(\mathsf{did},\mathsf{loc},\mathsf{mem})$: If $\mathsf{did} \in \mathcal{D}$ return $\perp$. Set $\mathcal{D}[\mathsf{did}] = (\mathsf{loc},\mathsf{mem})$.
$\mathsf{exec}(\mathsf{did},\mathsf{func}) \rightarrow out$: If $\mathsf{did} \notin \mathcal{D}$ return $\perp$. Compute func and return its output. func can read / write to $\mathcal{D}[\mathsf{did}].\mathsf{mem}$ or call any of $\mathcal{O}_{\mathsf{dev}}$ functions internally. If $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$ is called with input $\boldsymbol{\mu}$ and $\mathcal{D}[\mathsf{did}].\mathsf{loc} \in R^{\mathsf{in}}$, do $\mu^{\mathsf{rec}}.\mathsf{append}(\boldsymbol{\mu})$.
$\mathsf{sendTo}(\mathsf{did}_1,\mathsf{did}_2,\mathsf{data})$: If $\mathsf{data} \in \mathcal{D}[\mathsf{did}_1].\mathsf{mem}$, $\mathcal{D}[\mathsf{did}_2].\mathsf{mem}.\mathsf{append}(\mathsf{data})$.
$\mathsf{erase}(\mathsf{did},j)$: Erase index $j$, $\mathcal{D}[\mathsf{did}].\mathsf{mem}.\mathsf{erase}(j)$.
$\mathsf{seenInROU}(\mu^{\mathsf{all}})$: Return 1 if $\forall \boldsymbol{\mu} \in \mu^{\mathsf{all}}, \boldsymbol{\mu} \in \mu^{\mathsf{rec}}$ holds.
```

Figure 3: The device API.

3.4 Soundness

Our security definition for soundness involves two security experiments: a setup experiment and a challenge experiment. 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 geo-extraction, where we try to extract F using the Extract protocol. Geo-extraction is deemed successful only if F can be computed from a set of file fragments μ^{all} s.t. every fragment $\boldsymbol{\mu} \in \mu^{\mathsf{all}}$ was previously seen in a query to a commitment oracle associated with target region R^{in}. The definition says that the PoGeoRet scheme is sound if success in challenge experiment implies that geo-extraction succeeds.

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.

The adversary $\mathcal{A}_{\mathsf{setup}}$ may interact arbitrarily with the verifier; it may create files and cause the verifier to 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 purpose of the setup experiment is to run 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.

During geo-extraction, the same adversary component $\mathcal{A}_{\mathsf{chal}}$ is reused. Geo-extraction has three steps in total. First Extr.Derive derives file fragments μ^{all} from interactions with $\mathcal{A}_{\mathsf{chal}}$. We allow the adversary to be rewound in this step, as is standard in the PoRet literature, e.g., [31, 40]. 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^{in}. Geo-extraction succeeds only if step 2 and 3 both succeed. Finally a PoGeoRet scheme is said to be sound if adversarial success in the challenge experiment implies that geo-extraction succeeds w.h.p.

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

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

Figure 4: Setup and Challenge Experiments.

To infer whether a file fragment $\boldsymbol{\mu}$ is inside the target region R^{in}, we make use of a commitment oracle $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$. $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$ models the Commit function and records inputs (file fragments) whenever the oracle is invoked from a device located in R^{in}. So if $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$ was queried about a fragment, then it must have been inside R^{in}, and consequently if enough file fragments are inside R^{in}, then the file F itself is in R^{in}.

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^{in}, 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 recompute the file F from the file fragments input to $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$.

We present our detailed security experiments in Sec. 3.4.1. They come together in our soundness definition in Sec. 3.4.2.

3.4.1 Soundness security experiments.

We model device actions in our security experiments via the device oracle $\mathcal{O}_{\mathsf{dev}}$ specified in Fig. 3. $\mathcal{O}_{\mathsf{dev}}.\mathsf{init}$ is used to store the target region of uncertainty R^{rou}. $\mathcal{O}_{\mathsf{dev}}.\mathsf{createDevice}$ is used to spawn a device at a given location. $\mathcal{O}_{\mathsf{dev}}.\mathsf{exec}$ allows the adversary to execute a function func on a device of its choice, including any function in the PoGeoRet API. Within the API, PoGeoRet.Commit is explicitly modeled using the commitment oracle $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$, which tracks all calls to Commit. $\mathcal{O}_{\mathsf{dev}}.\mathsf{erase}$ allows erasing existing memory. Finally, $\mathcal{O}_{\mathsf{dev}}.\mathsf{seenInROU}$ is used to check if a given set of inputs were previously seen in a call to $\mathcal{O}^{\mathsf{in}}_{\mathsf{com}}$ inside R^{rou}.

Note that exec can also communicate with other devices through $\mathcal{O}_{\mathsf{dev}}.\mathsf{sendTo}$ (or) execute code on a different device. In all our experiments, the adversary is given complete freedom to call any device function. Both the experiments are in Fig. 4.

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

Challenge experiment $\mathbf{Exp}^{\mathsf{chal}}_{\mathcal{A}}$: In our second experiment, $\mathcal{A}_{\mathsf{chal}}$ responds to a Commit challenge issued by the verifier. The adversary is deemed successful if it generates a response that succeeds with a probability at least ϵ. Note that we issue one PoGeoRet challenge which internally comprises of one PoRet challenge. Success probability for the challenge experiment is defined as:

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

3.4.2 Soundness definition.

Our main security definition is:

[6] In practice, δ would have to be decided based on the city geography.

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 $\mathcal{A}$:*

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

It 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 p probability. 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., $p = 1 - \mathsf{negl}(\lambda)$. Note that the extraction algorithm Extr.Derive interacts with $\mathcal{A}_{\mathsf{chal}}$ initialized with the state s output by the setup experiment.

The above definitions of soundness and completeness assume an interactive protocol between the prover and verifier. In practice, non-interactive schemes are often desirable. We aim to build such a scheme in GoAT. Due to lack of space, we provide non-interactive definitions in App. B.1 (they only require minor modifications).

3.5 Practical model variants

There are two modeling assumptions which, by assuming an *economically rational* adversary, can lead to significantly better performance, and which we therefore embrace in GoAT. The first assumption—a lower bound on bandwidth costs—dictates when and how challenges may be issued to a provider, which in turn allows use of internet infrastructure for effective bootstrapping. The second assumption—rational behavior by an adversary in file-block retention—allows for use of fast (linear-time) erasure codes.

These two models are of independent interest beyond GoAT; for example, they could be applied to a PoRet scheme.

3.5.1 Challenge regimes.

In the previously described model, the verifier challenges the prover at random times. We refer to this challenge pattern as the *random-challenge model*. Building a practical PoGeoRet scheme under this model requires an existing network of verifiers, thereby posing a bootstrapping problem.

A more practical model, we believe, is one, based on existing internet infrastructure. In GoAT, we derive challenges from signatures provided by internet servers. But since the verifier is not issuing challenges, the prover decides when the challenge-response interaction is going to take place. We call this model the *flexible-challenge model*, signifying the extra flexibility prover has.

At first sight, such a model might not seem to work, as the prover can download the entire file before initiating the challenge-response interaction. But we argue that by imposing a restriction that challenge-response interactions take place once per interval, and by using a small interval length (i.e., high challenge frequency), the flexible-challenge model meets our security goals—assuming an economically rational adversary.

We now discuss the operational and security models for flexible-challenge model and end with an example.

Operational model: Time runs in *epochs*. Each epoch is in turn composed of I *intervals*, each of length β. Interval length determines challenge frequency, i.e., challenges are issued once per interval. Epoch length determines verification frequency, i.e., challenge responses are accumulated over the I intervals are generated and verified only at the end of an epoch.

Security model: Like before, we begin with a setup experiment where the adversary picks a file F and initializes 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 O_{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.

The model includes a bandwidth constraint: only ϕ bytes can be transferred from devices in R^{out} to those in R^{in} ($\phi \ll |F|$) per interval. The bound ϕ is meant to reflect the economics of storage: A rational adversary will not incur bandwidth costs in excess of the revenue it receives for storage. Bandwidth costs today are several orders of magnitude more than that of storage, as shown below.

Example: For the purpose of this example, we compare the bandwidth and storage costs charged by Amazon (the storage cost is used as a proxy for revenue). Let's say we set the interval length $\beta = 30$mins. If the encoded file size is $|F^*|$=1TB, then the storage revenue is at most \$0.02 per interval on Amazon S3 [11]. 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, and therefore $\phi = 1$GB; more broadly the relation between the bandwidth cap and the encoded file size is given by $\phi = \frac{|F^*|}{1000}$.

3.5.2 Rational file retention and erasure-coding.

Recall that a PoRet encoding F^* of file F incorporates an erasure code (to amplify soundness). To get strong soundness based on the security definitions we have given, it is essential to use a code with high distance between codewords—e.g., a maximum distance separable (MDS) code such as Reed-Solomon—treating F as a codeword. Such erasure coding is robust to adversarial erasures. MDS coding, however, is expensive in practice for large codewords, asymptotically at best $O(n\log n)$ for file size n (given tolerance of a constant-fraction of erasures) [35, 41], and very costly in practice. (To avoid this problem, a number of PoRet protocols, e.g., [20, 31], have "striped" files, i.e., broken them up into multiple codewords, permuting and encrypting error-coding symbols across codewords to tolerate adversarial erasures / corruption.)

A second, more practical approach, we believe, is to use weaker erasure codes, specifically fast (e.g., linear-encoding-time) codes, e.g., [42]. Such codes are far more performant for large files than MDS coding. They are designed, however, for noisy channels with random erasures, and are brittle in the face of adversarial erasures. Thus they provide poor security against a malicious adversary.

Given a *rational* adversary, however, it is possible to achieve good security with linear-time coding. Such an adversary may be viewed as seeking to maximize its financial gain and minimize its expenditure on storage. All other things being equal, however—for instance, given a certain amount of allocated storage in a given

[7] AWS bandwidth costs vary by region, ranging from \$20-\$100 per TB transferred [10]. S3 charges also vary by region, we use the maximum above.

geolocation—such an adversary will attempt to preserve F. We model this formally through a greedy storage algorithm that decides where to place bits of a given file in a greedy way with preference given to devices in R^{in}. The adversary still retains control of *storage capacities* of the devices, thus effectively deciding the amount of storage space inside R^{in}. More details are in App. B.2.

Assuming rationality in a provider reflects natural ecosystem design decisions. For example, a provider may be paid for retrieving F, or may earn a reputation for reliable service. Such a provider has a financial incentive to ensure that a stored file F is recoverable. The provider will not strategically erase file blocks in an attempt to render F unrecoverable if it has to store the same amount of data anyways. Consequently, it is possible to achieve strong soundness using a linear-time erasure code.

4 THE GOAT PROTOCOL

We now present details of our PoGeoRet scheme, GoAT. We begin by discussing GoAT-specific modeling details (Sec. 4.1). Next we provide a brief description of the Shacham-Waters (SW) PoRet scheme (Sec. 4.2). In Sec. 4.3, we present the two variants of GoAT: GoAT-H and GoAT-P, that use slightly different variants of the SW PoRet scheme. In Sec. 4.4, we discuss the security guarantees GoAT provides and introduce a new knowledge assumption. Finally in Sec. 4.5, we discuss extensions supporting use of low-resolution anchors and ways to decentralize trust among anchors.

4.1 System Model

We describe modeling details specific to GoAT now. GoAT achieves soundness under the flexible-challenge model. All the important notation is tabulated in Table 2.

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}. And the minimum speed required for storage providers joining our system is S_{min}. The ratio $\omega = S_{\mathsf{max}}/S_{\mathsf{min}}$ is the network speedup of the adversary. These parameter values need to be decided based on empirical measurements (See Sec. 5). Honest providers only need to attain the speed S_{min} for a short period of time. For example, if interval length $\beta = 1\text{hr}$, good connectivity for a few seconds every hour suffices.

We also include a small startup cost t_{start} as it dominates the round trip times for nearby locations. The expected maximum time for a round trip between two locations L_1 and L_2 is given by $\mathsf{rtt}_{\mathsf{max}}(L_1,L_2) = (2\mathsf{dist}(L_1,L_2)/S_{\mathsf{min}}) + t_{\mathsf{start}}$.

Anchors: GoAT leverages existing internet servers called anchors. Anchors must provide an authenticated time API and have a static known location. The time need not be absolutely correct, relatively consistent time is allowed. Clock drift is assumed to be negligible. To begin with, anchors are assumed to be honest. Decentralizing trust in anchors is discussed in Sec. 4.5.

Anchors serve time through an API denoted "GetAuthTime". It must take as input a nonce N and return a transcript $T = \{M,\sigma\}$ where $M = \{t,N\}$ is a message containing the time t and nonce (M could also contain other data), and σ is a signature over M, i.e., $\sigma = \mathsf{Sig}_{\mathsf{sk}_{\mathsf{A}}}(M)$. The key pair $(\mathsf{sk}_{\mathsf{A}},\mathsf{pk}_{\mathsf{A}})$ are the secret, public key of the anchor respectively. We assume a list of anchors $\mathcal{T}$ is decided based on various factors including which locations to support, anchor trustworthiness and reliability.

Notation	Description
ϵ	Frac. of queries answered correctly
ρ	Erasure code rate
I,β	#intervals per epoch, interval length
$R = (L;\delta)$	Circular region defined by center L & radius δ
ω	Network speedup
T	Anchor transcript
$\boldsymbol{\mu}$	File fragment
n, s, k	#blocks, #sectors-per-block, #challenges in SW
ϕ	Bandwidth cap
α	Grinding cap (2^{α} GetAuthTime calls)
a	Amplification factor

Table 2: Notation

The timestamp resolution of an anchor Γ_{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.

Storage model: We assume storage providers use SSDs for storage (we do not support HDDs, see [36]). Modern SSDs are quite fast with seek times of just a few milliseconds [8] due to inbuilt parallelism.

4.2 Shacham-Waters scheme

At a high level, SW uses BLS-style signatures to facilitate proof aggregation and public verifiability. We explain important details of the SW protocol [40] now omitting low-level details, which can be found in Fig. 9 in the Appendix.

Let $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$ be a computable bilinear map, with $\mathbb{Z}_p$ the support for $\mathbb{G}$. The setup in SW (SW.Setup) involves dividing the file into n blocks, with each block further divided into s sectors. Each sector is a symbol in $\mathbb{Z}_p$. For every block, an aggregate signature is computed over all the sectors. $\{\sigma_i\}_{i=1}^{n}$ denote the signatures.

Each challenge in SW.Chal consists of a block number $c_i \in [n]$ and a coefficient $v_i \in \mathbb{Z}_p$, both derived randomly. Denote the number of challenges by k. Generating a proof in SW.Prove requires computing a linear combination of the k file blocks to compute $\boldsymbol{\mu} = \left\{\mu_j\right\}_{j=1}^{s}$ (Bold face denotes vectors) and aggregating the corresponding signatures to compute σ. Crucially, the proof size does not depend on the number of challenges k.

Modifications to SW: We add a function SW.Commit that computes and commits the value $\boldsymbol{\mu}$ using a vector commitment (VC) scheme. Note that $\boldsymbol{\mu}$ is computed exactly as in the proof function.

We consider several variants of SW that differ in the choice of VC. The main one SW-P uses the *Correlated Pedersen scheme* where two normal Pedersen commitments are computed with correlated bases. That is, given bases $(\mathbf{h}_1, \mathbf{h}_2 = \mathbf{h}_1^{a})$ and a vector $\boldsymbol{\mu}$, compute $(\mathbf{h}_1^{\boldsymbol{\mu}}, \mathbf{h}_2^{\boldsymbol{\mu}})$ where the exponentiation operation between the two vectors denotes the Pedersen commitment computation. Another variant SW-H uses a hash function like SHA2 to commit. Referring back to the formalism from Sec. 3, the crucial Commit function in GoAT is vector commitment and $\boldsymbol{\mu}$ acts as the file fragment.

The verification function (SW.Verify) takes a commitment of $\boldsymbol{\mu}$ as an extra input and verifies that the value $\boldsymbol{\mu}$ provided in a proof matches the commitment. Other functions are unchanged.

4.3 GoAT protocols

We now present the GoAT protocol which relies on high-resolution anchors, i.e., an anchor A with millisecond or lower timestamp resolution (or) $\Gamma_{A} \leq 1$ms. A real-world example of such an anchor can make use of Roughtime [9], which has a 1µs resolution. Low-resolution anchors are considered in Sec. 4.5.1.

Two variants of GoAT arise depending on the PoRet scheme used. The first, GoAT-H, uses SW-H PoRet scheme, and admits a security proof in the random oracle model. The second, GoAT-P, uses SW-P PoRet scheme and is more performant than GoAT-H with at least 3x smaller proof sizes, but its security is based on a new assumption. The two schemes are largely similar, so we present GoAT-P first and discuss modifications needed for GoAT-H subsequently.

4.3.1 GoAT-P. As usual, our setting involves a user U that wants to store a file F with a provider situated in a location $L_{P} \in R^{\text{target}}$, a target region. Since GoAT is a non-interactive protocol, all APIs have the preamble NI. As noted before, the non-interactive PoGeoRet API is in the Appendix (Fig. 7).

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

As noted before, we assume that a set of anchors $\mathcal{T}$ is predetermined; let $A \in \mathcal{T}$ 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. Other protocol parameters such as the interval length β, number of intervals per epoch I are assumed to be predetermined.

Proof generation (NIProve**):** Generating a proof of geo-retrievability happens in two phases. In the first, geo-commitment generation phase, the provider interacts with the anchor 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 only once per epoch, the provider computes the full PoRets.

*Geo-commitment generation (*NIGeoCommit*):* The key idea is to sandwich the file access operation between successive pings to the anchor. The signature returned in the first ping is used as a PoRet challenge. A PoRet commitment is computed which is set as the nonce in the second ping. Fig. 5 depicts the geo-commit protocol explained now:

(1) *Ping #1:* Sample a random nonce N_1 and send a request GetAuthTime(N_1) to A. Receive transcript $T_1 = \{M_1,\sigma_1\}$ where $M_1 = \{t_1,N_1\}$ and $\sigma_1 = \text{Sig}_{\text{sk}_{A}}(M_1)$.
(2) *PoRet commitment:* Use σ_1 as randomness to derive a set of challenges $\mathcal{S} \leftarrow$ SW.Chal(η,pp,σ_1). Now generate a commitment com $\leftarrow$ SW-P.Commit(η,$\mathcal{S}$). (Refer to Sec. 4.2 (or Fig. 9) for a description of SW.Chal and SW-P.Commit.)
(3) *Ping #2:* Set nonce $N_2 = \text{com}$ and ping the anchor A again via GetAuthTime(N_2). Receive $T_2 = \{M_2,\sigma_2\}$ where $M_2 = \{t_2,N_2\}$.

We refer to the pair $C^{\text{geo}} = \{T_1,T_2\}$ 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 provider has I geo-commitments $\{C_m^{\text{geo}}\}_{m=1}^{I}$.

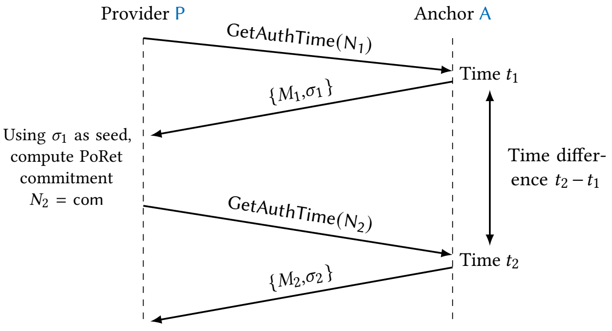

Figure 5: The geo-commitment protocol.

*PoRet computation (*NIPoRCompute*):* Once an epoch ends, the provider finishes proof generation by computing PoRets corresponding to the commitments computed during the epoch.

A naïve approach is to simply run the SW.Prove function I times with the same challenge sets used in step 2 of the geo-commit phase. But this leads to larger proofs. (Looking ahead, GoAT-H takes this naïve approach.)

Instead we aggregate proofs in much the same way as 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 $j \in \{1,\ldots,I\}$. 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 $\mathcal{S}^* = \cup_{j=1}^{I} \mathcal{S}_j^*$.

Intuitively, the set of coefficients $\{r_j\}$ ensures that a malicious provider cannot skip accessing the file even for a single interval. We give further details later.

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

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

Proof verification (NIVerify**):** The auditor checks anchor transcripts in the I geo-commitments using the anchor's public key. Then the auditor 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 $C^* = \prod_{j=1}^{I} (\text{com}_j)^{r_j}$. The proof of retrievability π^{PoRet} and C^* are verified by SW.Verify(pp,$\mathcal{S}^*$,C^*,π^{PoRet}).

Note that verification succeeds even with randomization of the challenge coefficients because SW-P.Commit contains only linear operations and the vector commitment scheme is homomorphic.

The final verification step is to check that the two timestamps are close in all geo-commitments, namely that $t_2 - t_1 \leq \Delta(L_{A},L_{P})$, where Δ is a pre-agreed upon function that takes anchor, provider locations as inputs and outputs the maximum runtime of NIGeoCommit operations (those happening between times t_1, t_2). We now discuss how Δ is set. Figure 12 in the Appendix specifies the GoAT protocol.

Setting Δ: Deciding Δ requires effectively 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 provider.

As shown in Fig. 5, the time difference $t_2 - t_1$ captures the time taken to run two operations: a 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 $\Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) = t_{\text{ping}} + t_{\text{com}}$.

Elapsed time for the GetAuthTime API call depends on the physical distance between the anchor and provider. We have $t_{\text{ping}} = \text{rtt}_{\max}(L_{\mathsf{A}}, L_{\mathsf{P}}) + t_{\text{proc}}$ where the first term denotes the maximum round trip time introduced in Sec. 4.1 and t_{proc} denotes the maximum processing time by the provider and anchor (processing time accounts for the time taken to compute a response, see Sec. 5.2 for details). Therefore we have:

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

We later prove that for a provider to succeed in a PoGeoRet proof for F, most of F must be stored within $\Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \cdot S_{\max}/2$ distance of the anchor location L_{A}.

Crucially, the radius of the ROU grows linearly with $\Delta(L_{\mathsf{A}}, L_{\mathsf{P}})$. This serves as a motivation to minimize computation time in a PoGeoRet as much as possible. Indeed, it is to reduce t_{com} that we introduce a commitment function as a means to commit to a PoRet proof before generating the proof itself.

Grinding attacks: Since NIGeoCommit 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 $p = 1 - (1 - g^k)^{2^{\alpha}}$. The adversary needs to choose the file-fraction g such that p is non-negligible, i.e., $g \geq (1 - (1 - 2^{-\lambda})^{2^{-\alpha}})^{1/k}$ (or) $g > 2^{\frac{-\lambda-\alpha}{k}}$ (via binomial expansion). Intuitively as the number of challenges k is raised, the adversary is forced to store more. We derive an exact constraint involving k and α in our security proofs.

Coefficient randomization: Randomization at the end of an epoch is necessary to ensure that the PoRet commitments $\{\text{com}_i\}$ 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., $\text{com}_j = (H_i (\text{com}_i)^{-1})^{\tau} H_j$. 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 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., [4].

4.3.2 GoAT-H.

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.

Geo-commitment generation (NIGeoCommit) is same as before except the change in the PoRet commitment function. 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 $\pi_j^{\text{PoRet}} \leftarrow \text{SW.Prove}(\eta, \mathcal{S}_j)$; the final proof is $\pi^{\text{PoRet}} \leftarrow \{\pi_1^{\text{PoRet}}, \pi_2^{\text{PoRet}}, \ldots, \pi_I^{\text{PoRet}}\}$. Accordingly, verification involves running SW.Verify I times.

The proof size and computation times in GoAT-H are same as GoAT-P asymptotically, but with higher constants (about 3x bigger proofs). Geolocation quality remains the same.

4.4 GoAT security

We discuss the security of GoAT-H and GoAT-P now. GoAT-H operates in the random oracle model and it's 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 introduce now, called the KEV Assumption (KEVA).

KEVA extends the commonly used KEA1 [14] for a vector of elements. It states that if $\mathcal{A}$ takes two correlated sets of bases $(\mathbf{h}_1, \mathbf{h}_2 = \mathbf{h}_1^a)$ as input and outputs (c_1, c_2) s.t. $c_2 = c_1^a$, 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 2 (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.*

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

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

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

GoAT-P achieves the same security as GoAT-H except that it requires the KEV assumption. The proof sketches are in App. D.

4.5 GoAT extensions

We discuss two extensions to GoAT: making GoAT work with TLS 1.2 anchors and decentralizing trust among anchors.

4.5.1 Low-resolution anchors (TLS 1.2). The NIGeoCommit protocol described before assumes anchors provide high-resolution time. But most existing anchors today such as TLS 1.2 servers only offer second-level resolution.

We deal with such anchors by *amplification*. The idea is to chain a sequence of proofs. Specifically, the prover alternates between computing a PoRet commitment and pinging the anchor, effectively filling an entire resolution tick this way. For example, TLS servers offer second-level resolution, so an entire second is filled with alternating PoRet commitments and anchor pings. The required length of the chain of proofs is set by the *amplification factor* a. a grows inversely with the expected time difference $\Delta(L_A,L_P)$; higher Δ means lower a and vice versa. More details can be found in App. A. In summary, amplification only causes a minor degradation in geolocation quality. But the proof size grows linearly with the amplification factor a.

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 groups. So we include a hash of the commitment and reveal the underlying commitment as part of the proof. Details can be found in App. A.

4.5.2 Decentralizing trust among anchors. It is straightforward to consider an extension to GoAT where as long as a threshold t number of anchors collude, the system is secure. This would come at the cost of somewhat more work to provers as they would have to execute NIGeoCommit with $t+1$ anchors every interval. But the proof size remains the same and the increase in prover / verifier computation time is not huge (See Sec. 5.2).

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 the more efficient variant of GoAT, GoAT-P, in approximately 2500 lines of C with support for both TLS 1.2 and Roughtime anchors. Our implementation uses TLSe [44] for TLS, Roughenough [43] for Roughtime and Relic [12] for pairings. We optimize the implementation of NIGeoCommit using the asynchronous I/O library libaio [6] and POSIX threads.

Section structure: In Sec. 5.1, we discuss a number of setup considerations, and in Sec. 5.2, we present our evaluation results.

5.1 Setup considerations

For the purposes of this paper, we only aim to demonstrate the feasibility of our approach. We thus set parameters conservatively, favoring strong completeness with somewhat looser geolocation bounds than may be achievable in practice. For more aggressive parametrization, a detailed internet measurement study is needed.

5.1.1 Network parameters. 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 [32].

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 [39], we set $S_{\mathsf{min}} = \frac{2}{9}c$, i.e., speedup $\omega = S_{\mathsf{max}}/S_{\mathsf{min}} = 3$ and the constant startup cost $t_{\mathsf{start}} = 5$ms. These parameter choices are consistent with recent work [19] that estimates the median RTT between PlanetLab nodes[8] and popular websites to be about 3.2× 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.

5.1.2 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.

In this study, we identify servers that return the correct time and have unique locations. We obtain server locations from an IP geolocation database, IP2Location.[9] We verify location uniqueness heuristically by finding each server's ISP and making sure it does not belong to a Content Distribution Network (CDN) [2]; servers that use CDNs do not have a fixed location since they respond from a replica closest to the query point. A stricter approach would be to perform a delay-based geolocation experiment validating that the server location is unique, e.g., [32, 46]. We do one such experiment for Roughtime on a small scale. For TLS 1.2 and Roughtime respectively, our findings are as follows.

TLS 1.2: We focus on domains belonging to 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 belong to a CDN provider. The result is a set of 300 domains that can be used as anchors, i.e., 10.5% of our original list. But this list is heavily biased towards anchors located in the U.S. (60% of the 300). So to find anchors for a different location, we apply more specific filters—e.g., to find anchors in UK we search for domains ending with ".ac.uk".

We also limit ourselves to using only those TLS 1.2 servers that use RSA for authentication. This is done purely for implementation convenience. We find that the proof transcript length for RSA-based servers is 389 bytes, which includes a 256-byte signature.

TLS uses TCP in the transport layer. Therefore in a standard TLS connection, it takes two round trips to get time: the first round trip establishes a TCP connection while the second gets the time. An important trick for better geolocation accuracy is to open all TCP connections prior to the start of the NIGeoCommit protocol. In our implementation, we open n sockets to the anchor in parallel if n pings are needed in NIGeoCommit.

Roughtime: We are aware of the existence of four Roughtime servers as of Apr. 2021. All of them return correct time with microsecond granularity. To check that their locations are unique, we perform a small geolocation experiment by sending an ICMP ping request from two vantage points: North Virginia (NV) and Singapore (SP). In this process, we identify one of the servers as unusable for geolocation, as it has a RTT of 17ms from NV and 30ms from SP, suggesting it is sitting behind a CDN provider. We find that the proof transcript length for Roughtime is 360 bytes.

[8] PlanetLab nodes tend to be well-connected to the internet, matching our expectation of storage provider's connection. [19] also picks geographically diverse nodes.

[9] These databases are known to have some errors [26] and a rigorous geolocation experiment like [46] would have to be done before deploying our system.

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 $t^{\text{tls}}_{\text{aproc}} = 6.5$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 $t^{\text{rt}}_{\text{aproc}} = 2$ms. This could be due to a combination of several factors, e.g., less load, faster transport layer (UDP) [19] and faster signature scheme (EdDSA).

5.1.3 GoAT parameters. We talk about how various parameters in GoAT are set now. App. C discusses some associated tradeoffs.

For SW PoRet, we use the BLS12-381 curve. Except in one experiment below, we set the number of sectors per block, $s = 96$.

As we discuss in Sec. 4.4, $(\rho + \frac{\phi}{|F^*|} + (1 - 2^{-(\lambda+\alpha)/k}))^k$ needs to be negligible. 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 $k = 250$;[10] note that the bandwidth cap is set to $\phi = 0.001|F^*|$ using the economic analysis from Sec. 3.5.1.

For the experiments below, we set the number of challenges $k = 100$. We expect minimal impact on results due to the slightly lower k.

Remaining parameters: In eq. (1), two more parameters remain to be set, t_{proc} and t_{com}. Note that we separate the processing time t_{proc} 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 $t_{\text{cproc}} = 1.5$ms and $t_{\text{com}} = 2$ms based on code benchmarks (the latter is discussed below).

5.2 Evaluation

We evaluate GoAT through several benchmarks and perform a real-world experiment over a week (Sec. 5.2.1). For most benchmarks, we use an AWS c5.4xlarge machine with 16 CPU, 32GB RAM and 2TB io2 SSD that is capped at 20k IOPS. The io2 SSD is only used for experiments with small duration as it is more expensive, whereas for the long experiment in Sec. 5.2.1, we use a 100 IOPS, 30GB gpt2 SSD. We do not expect this decision to have a significant impact as we show below that the effect of file sizes is negligible. 50 samples were taken in all experiments to compute the mean and standard deviation (shown 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 3 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 $t_{\text{com}} = 2$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.[11]

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)

Table 3: Time taken for PoRet commit with standard deviations.

#intervals	PoRetCompute (ms)	Verify (ms)
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 4: Computation time of PoRetCompute and Verify (vs) no. of intervals per epoch. Standard deviations in brackets.

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	$(s+1)e+2I\|T\|$

Table 5: GoAT proof sizes. e stands for the size of a single element in $\mathbb{G}$ or $\mathbb{Z}_p$ of the Shacham-Waters PoRet scheme.

Computation costs: Table 4 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.

In the above experiment, 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 first half of the equation (constant part) is contributed by the PoRet proofs, while the second half by anchor transcripts. If $I = 100$, GoAT-P proof size is 72.2KB with a dominating 70KB of anchor transcripts.

To show the dominant effect of anchor transcripts, we fix the interval length to $\beta = 1$hr and plot the proof size per interval as the epoch length is increased. The effect can be clearly seen in Fig. 6. The plot converges at 720B, the size of 2 Roughtime transcripts.

The proof sizes for all GoAT variants are in Table 5. Using same parameters as before ($I = 100$), if a TLS anchor (amplification $a = 20$) is used, the proof size of GoAT-P is 799.64KB. Whereas the proof size of GoAT-H is around 4.5MB for TLS anchor (about 5.7x bigger than GoAT-P) and 265.5KB for Roughtime anchor (about 3.6x bigger).

[10] Another set of parameters with higher code rate is $\rho = 0.5$, $k = 360$.

[11] Direct I/O (the "O_DIRECT" flag) is a way to avoid entire caching layer in the kernel and send the I/O directly to the disk.

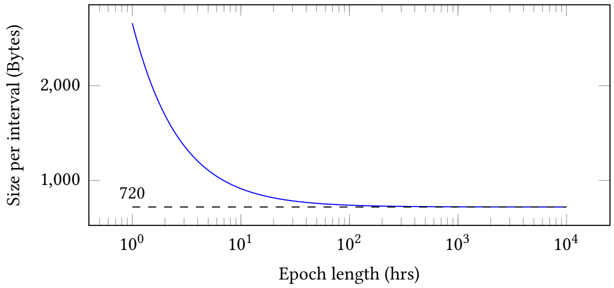

Figure 6: Proof size per interval of GoAT-P **with a Roughtime (RT) anchor against the epoch length. Interval length β = 1hr. Dashed line (720B) corresponds to the size of two RT transcripts.**

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 6: The ROU radius (δ) and the distance b/w anchor and closest AWS instance (d). All distances are in km. Last column shows the mean, standard deviation (SD) of the number of retries.

5.2.1 *Experiment.*

We devise a small experiment to demonstrate the practical feasibility of GoAT, specifically how it deals with network volatility. We focus on the NIGeoCommit protocol alone as it is the sole operation affected by network conditions. Prior works [29] have observed network stability over long time periods, and conclude that network instability is frequent but most often transient. So we handle failures in NIGeoCommit 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). Five anchors, screened for the criteria described above, are picked near each. The interval length is set to $\beta = 30$mins and the NIGeoCommit protocol is run for 10 days at NV (525 intervals) and 7 days at LON (347 intervals).

Table 6 shows the ten anchors used (the three Roughtime anchors are identifiable by their prefix). The first five anchors are used with the AWS instance in NV, the remaining with the one in LON. The 2nd column shows the distance d between the anchor and AWS instance (provider), the 3rd column shows the amplification factor a, and the 4th column shows the ROU radius δ along with the ratio δ/d. These three column values are computed as previously described.

Geolocation accuracy: The ROU radius to distance ratio δ/d is useful to understand the key factors contributing to the quality of geolocation. Recall from Sec. 4 that this ratio is given by $(\omega + ((t_{\text{com}} + t_{\text{start}} + t_{\text{proc}})S_{\max}/2\text{dist}(L_{\text{A}}, L_{\text{P}})))$. For providers farther away from the anchor, we see the ratio converging to speedup $\omega = 3$ suggesting that distance-to-the-anchor is the dominating factor. But for nearby providers, we see high ratios going up to 46; so the worse geolocation is caused by constants like t_{start}, t_{proc}.

As noted before, we choose parameters quite conservatively. If finer geolocation is desired, aggressive parametrization can help. For example, the variable t_{start} (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. C discusses various other optimization strategies.

TLS anchors achieve somewhat worse geolocation compared to Roughtime ones due to the higher processing times; for example, compare "american.edu" and "roughtime.chainpoint.org".

Robustness of our network model: The last column in Table 6 shows a statistical picture of the number of retries required to succeed during the experiment period. The anchor "holycross.ac.uk" behaved perfectly requiring no retries throughout. Whereas the anchor "www.sunysuffolk.edu" was the *only* one to fail—it failed in 4 of the 525 intervals, i.e., 0.7% false rejection. The four failures happened in consecutive intervals suggesting a period of bad server response times. We expect system designers to select several anchors in each location to avoid false rejections in practice. The maximum number of retries required to succeed was 13 (seen once with "sruc.ac.uk" and "nott.ac.uk").

6 RELATED WORK

A long line of works aim to prove correct file storage by a storage provider, e.g., Proof of Retrievability [31, 40], Proof of Data Possession [13, 27] and more recently Proof of Replication [16, 21, 24, 25].

To the best of our knowledge, only few works [18, 47] aim to prove file location. [18] 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). [47] 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 [47]. Digging deeper, this is due to our novel use of fast PoRet commitments whereas [47] naïvely combines the SW PoRet and PoL protocols. [47] also entrusts anchors with proof verification making the use of legacy anchors impossible unlike GoAT.

Most geolocation technologies in use today (e.g., GPS, Bluetooth beacons [30]) rely on trusted verifiers and are hence unusable in decentralized systems.

7 CONCLUSION

We have 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. We formalized the notion of PoGeoRet soundness by extraction from devices located within a geographic boundary. We also presented a few practical model variants that facilitate realization of GoAT. GoAT has a unique challenge model that permits batching proofs over several intervals and verifying them at the end of an epoch. GoAT proofs are small due to aggregation of PoRet proofs across the epoch. We have demonstrated GoAT's practicality through a fully functional implementation and a real-world experiment.

REFERENCES

[1] 2021. Alexa Top Sites. https://www.alexa.com/topsites. [Accessed Apr 2021].

[2] 2021. Content delivery network. https://en.wikipedia.org/wiki/Content_delivery_network. [Accessed Apr 2021].

[3] 2021. Cryptocurrency Prices by Market Cap. https://coinmarketcap.com/ [Accessed May 2021].

[4] 2021. Drand - Distributed Randomness Beacon. https://drand.love/ [Accessed May 2021].

[5] 2021. Filecoin Aims to Use Blockchain to Make Decentralized Storage Resilient and Hard to Censor. https://www.infoq.com/news/2021/02/filecoin-blockchain-storage/ [Accessed May 2021].

[6] 2021. Linux-native asynchronous I/O access library. https://pagure.io/libaio. [Accessed Apr 2021].

[7] 2021. Metric space. https://en.wikipedia.org/wiki/Metric_space. [Accessed Apr 2021].

[8] 2021. SSD UserBenchmarks - 1058 Solid State Drives Compared. https://ssd.userbenchmark.com/. [Accessed Apr 2021].

[9] A. Langley A. Malhotra and W. Ladd. 2020. Roughtime. https://datatracker.ietf.org/doc/html/draft-roughtime-aanchal.

[10] Amazon. 2021. AWS EC2 Costs. https://aws.amazon.com/ec2/pricing/on-demand/. [Accessed Apr 2021].

[11] Amazon. 2021. AWS S3. https://aws.amazon.com/s3/. [Accessed Apr 2021].

[12] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. [n.d.]. RELIC is an Efficient LIbrary for Cryptography. https://github.com/relic-toolkit/relic.

[13] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. 2007. Provable Data Possession at Untrusted Stores. In *ACM CCS*. 598–609.

[14] Mihir Bellare and Adriana Palacio. 2004. The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In *Annual International Cryptology Conference*. Springer, 273–289.

[15] J. Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* abs/1407.3561 (2014). arXiv:1407.3561 http://arxiv.org/abs/1407.3561

[16] Juan Benet, David Dalrymple, and Nicola Greco. 2017. Proof of replication. *Protocol Labs, July* 27 (2017), 20.

[17] J Benet and N Greco. 2018. Filecoin: A decentralized storage network. *Protoc. Labs* (2018), 1–36.

[18] Karyn Benson, Rafael Dowsley, and Hovav Shacham. 2011. Do you know where your cloud files are?. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. 73–82.

[19] Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. 2017. Why is the internet so slow?!. In *International Conference on Passive and Active Network Measurement*. Springer, 173–187.

[20] David Cash, Alptekin Küpçü, and Daniel Wichs. 2017. Dynamic proofs of retrievability via oblivious RAM. *Journal of Cryptology* 30, 1 (2017), 22–57.

[21] Ethan Cecchetti, Ben Fisch, Ian Miers, and Ari Juels. 2019. Pies: Public incompressible encodings for decentralized storage. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1351–1367.

[22] M. Clark. Mar 11, 2021. NFTs, explained. https://www.theverge.com/22310188/nft-explainer-what-is-blockchain-crypto-art-faq [Accessed Apr 2021].

[23] T. Dierks and E. Rescorla. 2008. TLS 1.2 RFC 5246. https://tools.ietf.org/html/rfc5246.

[24] Ben Fisch. 2018. PoReps: Proofs of Space on Useful Data. *IACR Cryptol. ePrint Arch.* 2018 (2018), 678.

[25] Ben Fisch, Joseph Bonneau, Nicola Greco, and Juan Benet. 2018. Scaling proof-of-replication for filecoin mining. *Benet//Technical report, Stanford University* (2018).

[26] Phillipa Gill, Yashar Ganjali, Bernard Wong, and David Lie. 2010. Dude, where's that IP? Circumventing measurement-based IP geolocation. In *Proceedings of the 19th USENIX conference on Security*. 16–16.

[27] Christian Hanser and Daniel Slamanig. 2013. Efficient simultaneous privately and publicly verifiable robust provable data possession from elliptic curves. In *2013 International Conference on Security and Cryptography (SECRYPT)*. IEEE, 1–12.

[28] Elizabeth (Liz) Harding, Lisa J. Acevedo, and Lindsay R. Dailey. 2021. Data Localization and Data Transfer Restrictions. https://www.natlawreview.com/article/data-localization-and-data-transfer-restrictions/. [Accessed Aug 2021].

[29] Toke Høiland-Jørgensen, Bengt Ahlgren, Per Hurtig, and Anna Brunstrom. 2016. Measuring latency variation in the internet. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 473–480.

[30] Kang Eun Jeon, James She, Perm Soonsawad, and Pai Chet Ng. 2018. 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

[31] Ari Juels and Burton S Kaliski Jr. 2007. PORs: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*. 584–597.

[32] Ethan Katz-Bassett, John P John, Arvind Krishnamurthy, David Wetherall, Thomas Anderson, and Yatin Chawathe. 2006. Towards IP geolocation using delay and topology measurements. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. 71–84.

[33] Protocol Labs. July 19, 2017. Filecoin: A Decentralized Storage Network. https://filecoin.io/filecoin.pdf. [Accessed Apr 2021].

[34] Storj Labs. October 30, 2018. Storj: A Decentralized Cloud Storage Network Framework. https://www.storj.io/storjv3.pdf. [Accessed Apr 2021].

[35] Sian-Jheng Lin, Tareq Y Al-Naffouri, Yunghsiang S Han, and Wei-Ho Chung. 2016. Novel polynomial basis with fast Fourier transform and its application to Reed–Solomon erasure codes. *IEEE Transactions on Information Theory* 62, 11 (2016), 6284–6299.

[36] Chris Mellor. January 25, 2021. 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/. [Accessed Apr 2021].

[37] Christopher Patton. 2018. Roughtime: Securing Time with Digital Signatures. https://blog.cloudflare.com/roughtime/. [Accessed Apr 2021].

[38] Qualys. April 11, 2021. SSL Pulse. https://www.ssllabs.com/ssl-pulse/. [Accessed Apr 2021].

[39] Paul Reinheimer and Will Roberts. [n.d.]. Global Ping Statistics ⟶ Manhattan. https://wondernetwork.com/pings/Manhattan. [Accessed Apr 2021].

[40] Hovav Shacham and Brent Waters. 2008. Compact proofs of retrievability. In *International conference on the theory and application of cryptology and information security*. Springer, 90–107.

[41] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. 2013. Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 325–336.

[42] Amin Shokrollahi. 2006. Raptor codes. *IEEE transactions on information theory* 52, 6 (2006), 2551–2567.

[43] Stuart Stock. 2021. Roughenough. https://github.com/int08h/roughenough. [Accessed Apr 2021].

[44] Eduard Suica. 2021. Single C file TLS 1.2/1.3 implementation. https://github.com/eduardsui/tlse/. [Accessed Apr 2021].

[45] D. Vorick and L. Champine. November 29, 2014. Sia: Simple Decentralized Storage. https://sia.tech/sia.pdf. [Accessed Apr 2021].

[46] Yong Wang, Daniel Burgener, Marcel Flores, Aleksandar Kuzmanovic, and Cheng Huang. 2011. Towards Street-Level Client-Independent IP Geolocation.. In *NSDI*, Vol. 11. 27–27.

[47] Gaven J Watson, Reihaneh Safavi-Naini, Mohsen Alimomeni, Michael E Locasto, and Shivaramakrishnan Narayan. 2012. Lost: location based storage. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*. 59–70.

Non-interactive Proof of Geo-Retrievability

- (sk,pk) ← NIKGen(1^λ): Generate key pair. Run by the user.
- (F^*,η,pp) ← NISetup(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.
- π^{geo} ← NIProve(η, 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^{geo} ← NIGeoCommit(η, 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 com ← NICommit($\boldsymbol{\mu}$) is a sub-function of NIGeoCommit which takes a file fragment $\boldsymbol{\mu}$ as input and generates a commitment com.
  - π^{PoRet} ← NIPoRCompute(η,$\mathcal{S}$): On input file handle η and a set of PoRet challenges $\mathcal{S}$, compute one or more proofs of retrievability.
- 0/1 ← NIVerify(pp,R,c,π^{geo}): The verifier checks that the file is in the desired region R by verifying the proof π^{geo} using the challenge, public params.

Figure 7: NIPoGeoRet API.

A SUPPORTING TLS 1.2 ANCHORS

A.1 Low-resultion 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

Geo-commitment generation (NIGeoCommit) between P and A

Prot_{A}: Follow the standard protocol (TLS 1.2 / Roughtime).

Prot_{P}: On input $\{\eta,\text{pp}\}$, 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 $\{t_i,N_i,\sigma_i\} \leftarrow \mathsf{A}.\text{GetAuthTime}(\text{N}_\text{i})$. If $i=a+1$, then break.
(2) (**PoRet commitment**) Run $\mathcal{S}_i \leftarrow \text{PoRet.Chal}(\eta, \text{pp}, \sigma_i)$, $\text{com}_i \leftarrow \text{PoRet.Commit}(\eta,\mathcal{S}_i)$. Set $N_{i+1}=\text{com}_i$, $i=i+1$ and repeat from step 1.

P saves $C^{\text{geo}} = \{T_i\}_{i=1}^{a+1}$ 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 $C^{\text{geo}} = \left\{ \{T_i\}_{i=1}^{a+1}, \{\text{com}_i\}_{i=1}^{a} \right\}$, the auditor V does:

- Set $N_1=\text{seed}$ and $\forall i \in [1,\ldots,a], N_{i+1}=\text{com}_i$.
- Verify anchor signatures using pk_{A}, $\forall i \in [1,\ldots,a+1]$, $\text{Verify}_{\text{pk}_{\mathsf{A}}}(\sigma_i, \{t_i,N_i\})$ 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 $t_2-t_1 \leq \Delta(\mathsf{A},\mathsf{P})$.

Figure 8: Geo-commitment protocols.

Shacham-Waters PoRet scheme

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

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

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

Figure 9: The Shacham-Waters PoRet schemes with an extra commitment step. SW-H, SW-P differ in the choice of VC scheme.

that this modification applies to both GoAT variants, GoAT-H and GoAT-P.

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 $a=\lfloor \Gamma_{\mathsf{A}}/\Delta(L_{\mathsf{A}},L_{\mathsf{P}}) \rfloor - 1$.[12] Below, we explain how to time proof execution in order to ensure receipt of $a+1$ transcripts with the same timestamp.

Proof of Retrievability

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

Figure 10: Publicly verifiable PoRet API. PoRet.Commit is the only addition compared to prior modeling [31].

Correlated Pedersen commitment CPVC

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

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

Hash-based commitment HVC

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

Figure 11: Pedersen and Hash-based Vector Commitment scheme

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 NIGeoCommit protocol for *any* anchor, low- or high-resolution, is specified in Fig. 8, 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 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

[12] 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.

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.

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

- $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{KGen}(1^\lambda)$: U runs $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{PoRet.KGen}(1^\lambda)$.
- $(F^*,\eta,\mathsf{pp}) \leftarrow \mathsf{NISetup}(\mathsf{sk},\mathsf{pk},F)$: U runs $(F^*,\eta,\mathsf{pp}) \leftarrow \mathsf{PoRet.Setup}(\mathsf{sk},\mathsf{pk},F)$ and picks a geographic region $R = (L;\delta_L)$. Values $\{F^*,\mathsf{pp},R\}$ are given to P situated at L.
- $C^{\mathsf{geo}} \leftarrow \mathsf{NIGeoCommit}(\eta, R)$: P selects an anchor $\mathsf{A} \in \mathcal{T}$ based on the input region R and generates geo-commitments via NIGeoCommit (See Fig. 8) once every interval until the end of epoch. Note that $\mathsf{NICommit}(\boldsymbol{\mu})$ = $\mathsf{PoRet.VC.Commit}(\boldsymbol{\mu})$.
- $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{NIPoRCompute}(\eta,\mathcal{S})$: Let $\mathcal{S} = \{\mathcal{S}_j\}_{j=1}^N$ denote the PoRet challenge sets derived in NIGeoCommit. 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 $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{SW\text{-}P.Prove}(\eta,\mathcal{S}^*)$.
- $\pi^{\mathsf{geo}} \leftarrow \mathsf{NIProve}(\eta,R)$: The two sub-functions of NIProve are specified above. The proof of geo-retrievability π^{geo} consists of geo-commitments output by NIGeoCommit and the proof(s) of retrievability output by NIPoRCompute.
- $0/1 \leftarrow \mathsf{NIVerify}(\mathsf{pp}, R, \pi^{\mathsf{geo}})$: 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. 8. Then the proof of retrievability π^{PoRet} is verified using the steps described below.
  - **SW-H**: Check $\forall j$, $\mathsf{SW\text{-}H.Verify}(\mathsf{pp}, \mathsf{c}_j, \mathsf{com}_j, \pi_j^{\mathsf{PoRet}}) = 1$ 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 $\mathsf{SW\text{-}P.Verify}(\mathsf{pp},\mathcal{S}^*,C,\pi^{\mathsf{PoRet}}) = 1$.

Figure 12: 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.

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 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 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 $\Delta(L_{\mathsf{A}},L_{\mathsf{P}}) = 50$ms and $\Gamma_{\mathsf{A}} = 1000$ms, then $e = 50/19 = 2.6$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:

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

A.2 Changes to Commit

As noted in the main body, the PoRet commitments in GoAT-P won't fit into the TLS nonce field. 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 Commit 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.

B FORMALISM EXTENSIONS

B.1 Non-interactive Proofs of Geo-Retrievability

Non-interactive Proofs of Geo-Retrievability or 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. 7) 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., NIProve and NIVerify.

Relation to GoAT: Recall that GoAT is a non-interactive protocol. So the API in Fig. 7 map to the GoAT protocol specified in Fig. 12.

For ease of explaining GoAT, we divide NIProve into two sub-functions, NIGeoCommit and NIPoRCompute. The former specifies the interaction with anchor A to derive challenges. NIGeoCommit for GoAT is specified in Fig. 8.

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

B.2 Rational file retention

We expand on the rational file retention assumptions made in Sec. 3.5.2 now. The device API is largely same as in Fig. 3 except a change to the createDevice function. The greedy storage algorithm greedyFill is in Fig. 13.

```
Modified device oracle $O_{\text{dev}}$
State: A region $R^{\text{in}}$. Key-value pairs $\mathcal{D}[\text{did}] = (\text{loc}, \text{cap}, \text{mem})$ where the key did is the device identifier, loc is its location, cap is the memory capacity and mem is a list denoting the memory. A list $\mu^{\text{rec}}$ to track inputs to the commitment oracle $O^{\text{in}}_{\text{com}}$.
init($R$): Set $R^{\text{in}} = R$. Not callable by the adversary.
createDevice(did,loc,cap): If did $\in \mathcal{D}$ return $\perp$. Set $\mathcal{D}[\text{did}] = (\text{loc,cap,\_})$.
exec(did,func) $\rightarrow out$: If did $\notin \mathcal{D}$ return $\perp$. Compute func and return its output. func can read / write to $\mathcal{D}[\text{did}]$.mem or call any of $O_{\text{dev}}$ functions internally. If $O^{\text{in}}_{\text{com}}$ is called with input $\boldsymbol{\mu}$ and $\mathcal{D}[\text{did}].\text{loc} \in R^{\text{in}}$, do $\mu^{\text{rec}}$.append($\boldsymbol{\mu}$).
sendTo($\text{did}_1$,$\text{did}_2$,data): If data $\in \mathcal{D}[\text{did}_1]$.mem, $\mathcal{D}[\text{did}_2]$.mem.append(data).
erase(did,$j$): Erase index $j$, $\mathcal{D}[\text{did}]$.mem.erase($j$). Do $\mathcal{D}[\text{did}]$.cap$-=1$.
seenInROU($\mu^{\text{all}}$): Return 1 if $\forall \boldsymbol{\mu} \in \mu^{\text{all}}, \boldsymbol{\mu} \in \mu^{\text{rec}}$ holds.
```

```
Greedy allocation algorithm greedyFill(mem)
$cur = 0$
for $R$ in $\{R^{\text{in}}, R^{\text{out}}\}$
  for did in $\mathcal{D}$
    $available = \mathcal{D}[\text{did}].\text{cap} - |\mathcal{D}[\text{did}].\text{mem}|$
    if $available \leq 0$ continue
    if $\mathcal{D}[\text{did}].\text{loc} \in R$ then
      $\mathcal{D}[\text{did}]$.mem.append(mem[$cur : cur + available$])
      $cur += available$
    endif
  endfor
endfor
```

Figure 13: The modified device API.

#sectors	Time (ms)
64	0.89 (0.01)
128	1.53 (0.01)
256	2.99 (0.01)

Table 7: Time taken for SW PoRet commit without the file read operation as a function of the number of sectors. Averaged over 10 runs.

Note that the memory is composed of many blocks. The size of each block is dependent on the protocol. For the GoAT protocol, this should be the same as the block size used in the PoRet so that all bits in a block are in the same device.

C PRACTICAL CONSIDERATIONS AND FUTURE WORK

C.0.1 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. 5.1.3, 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.

C.0.2 Anchor clocks. For GoAT to work, we assume that the clock drift of anchors is negligible. This assumption was made to ensure that clock offsets can be observed once and used later on, avoiding the need for any time synchronization. Clock drifts in practice tend to be much smaller than the interval lengths in GoAT, and hence this assumption is reasonable.

C.0.3 Finding anchors. In Sec. 5, we used just the basic requirements in deciding whether a given internet server can be used as an anchor. In practice though, other considerations such as reliability (does the anchor have stable response times) and trustworthiness (is the anchor reputable enough) will have to be taken into account. As noted before, if relying on existing internet servers is undesirable, anchors for GoAT can be purpose-built.

C.0.4 Optimizations 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_{start} (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_{start} could be used. More experiments to understand if this idea can be used in practice are needed.

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.

C.0.5 Constructing a proof-of-space. One potentially impactful research direction is to extend GoAT to construct a Proof-of-Space. Currently, GoAT can only prove that a file F is geographically retrievable from a set of different regions. But if the file F is adversarially chosen, the prover might only actually need to store a small seed.

D SECURITY PROOFS

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 GoAT-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., the PoRet verification protocol (PoRet.Verify) part of Verify must detect otherwise.
(2) A combination of timing and knowledge based arguments to prove that the Commit 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 $O^{\text{in}}_{\text{com}}$.
(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 $O^{\text{in}}_{\text{com}}$ 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 Commit on valid file fragments. This relies on the KEV assumption (See Def. 2) for GoAT-P and the ROM for GoAT-H. Next we argue that all calls to Commit 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 ($O^{\text{in}}_{\text{com}}$) 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. We begin with the high resolution 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.

D.1.1 High-resolution anchors ($a=1$). Fixing some notation, assume that the storage provider P is at a location $L_p \in R^{\text{target}}$ 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 $\delta=\Delta(L_p,L_1)\cdot S_{\max}/2$, i.e., the region $R^{\text{in}}=(L_1;\delta)$. Expanding the radius further we have, $\delta=(t_{\text{com}}+\text{rtt}_{\max}(L_p,L_1)+t_{\text{proc}})\cdot(S_{\max}/2)$.

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 D_{out} situated at $L_2 \in R^{\text{out}}$ on which one of the PoRet commitments is computed. By definition we have $\text{dist}(L_1,L_2)>\delta$.

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 D_{out}, and therefore the time taken to compute commitment on D_{out} is negligible, i.e., $t^{\mathcal{A}}_{\text{com}}=0$. We also set the anchor processing time $t^{\mathcal{A}}_{\text{proc}}=0$.

The time taken to receive and respond from D_{out} during the geo-commitment protocol with A is given by $z=2\text{dist}(L_1,L_2)/S_{\max}$. This is because in Fig. 8 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 D_{out}. Moreover, we do not include any startup cost when the adversary is sending messages between devices, so $t^{\mathcal{A}}_{\text{start}}=0$.

For the geo-commitment verification to succeed, it must be that $z \le \Delta(L_p,L_1)$. (See last step in Fig. 5 when $a=1$.)

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

D.1.2 Low-resolution anchors ($a>1$). The target region now has a slightly larger radius, $\delta=(\Gamma/a)\cdot S_{\max}/2$. 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 D_{out} (D_{out} is setup in the same fashion as before).

The time difference between last and first timestamp in NIGeoCommit is given by $z=2a\text{dist}(L_1,L_2)/S_{\max}$. 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 $z \le \Gamma$. Intuitively, this corresponds to $a+1$ timestamps having the same time. But we have a contradiction, as z must also satisfy $z>2a\delta/S_{\max}$, 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 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 $z = 2\text{dist}(L_1,L_2)/S_{\max}$ (incurred as the adversary would have to forward challenges required to compute com_1 from D_{in} to $\mathsf{D}_{\mathsf{out}}$).

D.2 Remaining proofs

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

D.2.1 Part-one proof. For this we reuse the proof for Theorem 4.2 in [40]. 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 $\{\mu_j\},\sigma$ 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.

D.2.2 Part-three proof. We re-purpose the extraction algorithm provided in the proof of Theorem 4.3 in [40]. [40] 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 $\epsilon - (\rho n)^k/(n-k+1)^k$ is positive and non-negligible.

Recall that our extraction algorithm Extract is composed of Extr.Derive and Extr.Assemble. And the extraction algorithm of [40] already follows this additional 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., $O_{\mathsf{dev}}.\mathsf{seenInROU}(\mu_i^{\mathsf{all}}) = 1\ \forall i \in \{1,2,\ldots,I\}$ 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 $g = (1-(1-2^{-\lambda})^{1/\alpha})^{1/k}$ 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 [40] 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 $w = (\rho n)^k/(n-k+1)^k$ (omitting the negligible part of the equation caused by type-2 queries). The extractor needs $\rho n \le n$ 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 $w = (\rho n + \gamma)^k/(n-k+1)^k$. And assuming $\rho n + \gamma \le n$, extraction happens in $O(n/(\epsilon - w))$ time, i.e., same order as before. One constraint we get is $\frac{\phi}{|F^*|} \le g - \rho$.

We want $\epsilon - w$ to be positive and non-negligible. Therefore w needs to be negligibly small. Meaning $(\rho + \gamma/n)^k$ (or) $(\rho + \frac{\phi}{|F^*|} + 1 - g)^k$ needs to be negligible. As noted above, the number of interactions required and the time to extract is the same order as in [40]. □

Note that the above proof assumed that any ρ fraction of blocks can be used to extract the file F. This is not true for fast-codes (Sec. 3.5.2). But since we assume that the adversary only picks how many blocks to delete, and does not resort to strategic deletion of blocks, we can trivially extend the above proof to the setting of fast-codes by assuming that the adversary deletes blocks randomly. Please refer to Sec. 3.5.2 and App. B.2 for a discussion about this assumption.