GoAT: File Geolocation via Anchor Timestamping

Deepak Maram
Cornell Tech
sm2686@cornell.edu

Iddo Bentov
Cornell Tech
iddobentov@cornell.edu

Mahimna Kelkar
Cornell Tech
mahimna@cs.cornell.edu

Ari Juels
Cornell Tech
juels@cornell.edu

***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 rely on trusted-verifiers, GoAT uses public timestamping servers on the internet as geolocation anchors, tolerating a local threshold of dishonest 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 also 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.

I. 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) [20] 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 [13], Filecoin [32], Sia [45], Storj [33], 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 [15] uses a protocol called Proof of Replication (PoRep) [23] 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.

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 other settings. For example it can help prove compliance with laws specifying localized storage of certain forms of data, e.g., [27].[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

[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 national laws only require that a copy of data be stored locally whereas more stricter laws make transferring data abroad illegal [27]. The latter would require the use of trusted hardware together with techniques in this paper.

applications such as file replication.

We introduce a formal definition of PoGeoRets in this paper, and propose, implement, and experimentally validate a PoGeoRet system called GoAT. GoAT creates publicly verifiable file-replica geolocation proofs. GoAT proofs can thus be consumed by a multiplicity of verifiers and can be used to construct a system 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 [31], [46] and files [16], [47]—but make strong assumptions, e.g., all verifiers are honest, verifiers are close to storage 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.

A. *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. As we show, 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*.

B. *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. 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. We specify the adversarial model precisely later.

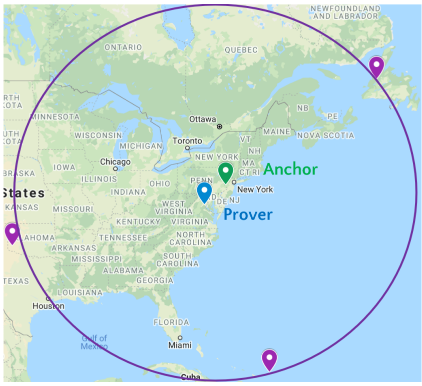

Fig. 1. A prover and a Roughtime anchor are situated 300km apart. GoAT's region of uncertainty (ROU) is a circle of radius 2000km shown in purple proving that the replica is stored in the east half of North America.

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 a honest prover to prove file replica possession successfully.

C. *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) [30], [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 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 it by as much as 75x 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 model of location-specific *storage devices*. A PoGeoRet is said to be sound if extraction succeeds from devices located in the desired target geographic region.

D. Contributions and Paper Organization

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

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. III). 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. IV. 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 the security of GoAT in App. C.
3) *Implementation and Evaluation*: To demonstrate GoAT's practicality we prototype GoAT 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. V).

We present related work in Sec. VI and conclude in Sec. VII. We have released GoAT as an open-source tool at https://github.com/GoATTeam/GoAT.

II. Preliminaries

A. 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.

1) TLS 1.2: Some TLS 1.2 servers [22] 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) 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.

B. Proof of Retrievability

Proof of Retrievability [30] 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 [30] and Shacham-Waters (SW) [40] are the two main choices.

Figure 8 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, e.g., on a blockchain. It enables any party to verify a proof of retrievability.

An 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 (ideally within a few milliseconds), and (2) compact. We specify our construction of PoRet.Commit later in the paper. The PoRet.Commit function is the only addition we make to the PoRet scheme used in GoAT, which otherwise remains unmodified.

Proof of Geo-Retrievability

- $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{KGen}(1^\lambda)$: Generate key pair. Run by the user.
- $(F^*,\eta,\mathsf{pp}) \leftarrow \mathsf{St}(\mathsf{sk},\mathsf{pk},F)$: 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.
- $c \leftarrow \mathsf{Chal}(\eta,\mathsf{pp},\mathsf{seed})$: On input file handle η and params pp, derive a challenge c from the input seed.
- $\pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta,R,c,\mathsf{pp})$: On input processed file η, a geographic region R and a challenge c, generates a proof of geo-retrievability π^{geo}. Run by the prover.
- $0/1 \leftarrow \mathsf{Verify}(\mathsf{pp},R,c,\pi^{\mathsf{geo}})$: The verifier checks that the file is in the desired region R by verifying the proof π^{geo} using the challenge, public params.

Fig. 2. API of a Proof of Geo-Retrievability (PoGeoRet) scheme.

III. FORMALIZING PROOFS OF GEOGRAPHIC RETRIEVABILITY

A Proof-of-Geographic-Retrievability (PoGeoRet) scheme includes three parties[3]: 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. III-A, explaining how a PoGeoRet leverages an underlying PoRet. We provide the adversarial model in Sec. III-B, and then presenting the basic modeling behind our formal definitions. We formalize completeness in Sec. III-C and soundness in Sec. III-D. Finally, in Sec. III-E2, we discuss modifications to our security model and definitions that we believe reflects real-world use cases such as support for fast file encoding and easy bootstrapping.

A. Preliminaries

Protocol structure: The API for a PoGeoRet scheme is specified 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 St to denote PoGeoRet.St.

We define a PoGeoRet for a general setting in which a target file F is stored as a publicly accessible plaintext. (As noted above, some decentralized file systems rely on file encryption by a file owner in order to achieve redundancy proofs.)

A user U that wants a file F to be stored near a particular location runs the setup protocol (PoGeoRet.St) on F to generate an encoded file F^*. Here, $\mathsf{St} = \mathsf{PoRet.St}$, i.e., we invoke the underlying PoRet setup to encode 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.

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[4] 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 $\mathsf{dist}(L,L') \leq \delta$. 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.

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. For simplicity, we assume that the attained ROU is same for all locations in the target region R^{target}.

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 $\delta = 10\text{km}$.[5] 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 make the notion of file presence within a geographic region precise, 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 they remain fixed throughout the experiment.

Devices have access to two kinds of memory: static and dynamic. The static memory is "frozen" after initialization,

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

[4] 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.

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

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

TABLE I
SYSTEM ENTITIES. ANCHORS ARE SPECIFIC TO GoAT.

i.e., it cannot be modified / erased during the security experiment, while the dynamic memory is modifiable throughout. Intuitively, the static memory corresponds to what a real-world prover plans to store in a given location most of the time.

Formally, we model all devices by way of an oracle $\mathcal{O}_{\mathsf{dev}}$ presented in Sec. III-D.

Modeling time: 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 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.

B. Adversarial Model

Table I 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 a honest-majority assumption for auditors).

We also assume that the adversary is rational. We explain what this means precisely in the security experiments below.

C. 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 $\mathsf{St}(\mathsf{sk}, \mathsf{pk}, F)$, the verification algorithm accepts when interacting with a valid prover, i.e., $\mathtt{Succ} \leftarrow \mathsf{Verify}(\mathsf{pp}, R^{\mathsf{rou}}, \pi^{\mathsf{geo}})$, for a proof π^{geo} generated by a prover located in the target region R^{target} and running, $\pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta, R^{\mathsf{rou}}, \mathsf{pp})$. For convenience, we hide challenges exchanged between prover and verifier in the above expressions. Since completeness is relatively straightforward, we omit further details.

D. Soundness

Our security definition for soundness involves three security experiments: a setup experiment, a challenge experiment, and an extraction experiment. The setup experiment lets the adversary set up its devices and pick a file F for the challenge-response interactions in the remaining experiments. The challenge experiment corresponds to interactions with a real-world verifier, and is used to ensure that an adversary responds to ϵ-fraction of queries correctly. Finally in the extraction experiment, we try to extract F from the static memory of devices in R^{in}. The definition then says that the PoGeoRet scheme is sound if the extraction experiment succeeds.

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

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

The purpose of the setup experiment is to run St on a file F picked by the adversary. The resultant output F^* is challenged in the challenge and extraction experiments.

Both challenge and extraction experiments involve performing only one function, receive a challenge and return a response. The difference between the two lies in how challenges are issued. During the challenge experiment, the challenges are issued to the second adversary component $\mathcal{A}$("chal"). $\mathcal{A}$("chal") has no restrictions; it is allowed to freely communicate messages between devices and use them in computations.

On the other hand, during the extraction experiment, we define an extraction algorithm Extract that uses data stored in the static memory of devices in R^{in} to compute challenge responses. Note that Extract operates directly on the device memory for reasons explained below.

Through the challenge experiment, we fulfill the goal of requiring the adversary to answer ϵ fraction of queries correctly. Whereas through the extraction experiment, we model the desired goal of file geolocation, i.e., if the extraction experiment succeeds, then the file F must have always been within R^{in} (as it must be in the static memory).

Our modeling of extraction directly from the devices i.e., without the adversary, corresponds to the expectation of rational behavior from prover in the real-world. That is, we assume that if the prover has access to the challenged file blocks, then the prover might as well respond to the challenges correctly. Moreover, in practice such behavior can be incentivized by attaching (say) monetary payments to valid PoGeoRet responses.

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 that the file is in the static memory of devices in R^{in}, but it isn't. Thus the aim of $\mathcal{A}$("setup") is to setup devices in a way that: (1) V accepts responses from $\mathcal{A}$("chal") with high probability and (2) V fails to extract the file via Extract.

We consider a stateful adversary. The state transfer between experiments is modeled implicitly through the storage devices.

We present our detailed security experiments in Sec. III-D1. They come together in our soundness definition in Sec. III-D2.

1) Soundness security experiments: We model device actions in our three security experiments via the device oracle $\mathcal{O}_{\mathsf{dev}}$ specified in Fig. 3. The entire PoGeoRet API (Fig. 2) is modeled using another oracle $\mathcal{O}^*$. The adversary is given access to $\mathcal{O}^*$ indirectly through $\mathcal{O}_{\mathsf{dev}}$ — the `execAny` and `execStatic` functions allow adversary to execute any function fn, including one of $\mathcal{O}^*$. The `execStatic` function is constrained to use just the static memory, while `execAny` has

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

```
State: A region $R^{\mathsf{in}}$. Key-value pairs $\mathcal{D}[\mathsf{id}] = (\mathsf{loc}, \mathsf{M_{dyn}}, \mathsf{M_{sta}})$ where the key id is the device identifier, loc is its location, $\mathsf{M_{dyn}}$ is the dynamic memory and $\mathsf{M_{sta}}$ is the static memory respectively. $\mathsf{M_{dyn}}$ is a list.
init($R$): Set $R^{\mathsf{in}} = R$. Not callable by the adversary.
create(id,loc,$\mathsf{M_{dyn}}$,$\mathsf{M_{sta}}$): If $\mathsf{id} \in \mathcal{D}$ return $\perp$. Set $\mathcal{D}[i] = (\mathsf{loc},\mathsf{M_{dyn}},\mathsf{M_{sta}})$.
execAny(id,fn) $\to out$: If $\mathsf{id} \notin \mathcal{D}$ return $\perp$. Compute fn using static / dynamic memory of device $\mathcal{D}[\mathsf{id}]$ and return its output.
execStatic(id,fn) $\to out$: If $\mathsf{id} \notin \mathcal{D}$ return $\perp$. Compute fn using just the static memory of device $\mathcal{D}[\mathsf{id}]$ and return its output.
dataTransfer($\mathsf{id}_1$, $\mathsf{id}_2$, $d$): If $d \notin \mathcal{D}[\mathsf{id}_1].\mathsf{M_{dyn}}$ and $d \notin \mathcal{D}[\mathsf{id}_1].\mathsf{M_{sta}}$ return $\perp$. Run $\mathcal{D}[\mathsf{id}_2].\mathsf{M_{dyn}}.\mathsf{append}(d)$.
erase(id,$j$): Erase index $j$, $\mathcal{D}[\mathsf{id}].\mathsf{M_{dyn}}.\mathsf{erase}(j)$.
clearDynamic(): For all $\mathsf{id} \in \mathcal{D}$, if $\mathcal{D}[\mathsf{id}].\mathsf{loc} \in R^{\mathsf{in}}$ set $\mathcal{D}[\mathsf{id}].\mathsf{M_{dyn}} = \phi$.
```

Fig. 3. The device API.

no such constraints. In the setup and challenge experiments, the adversary is given complete freedom to call any device function, modeled concisely as $\mathcal{O}^*_{\mathsf{dev}}$.

Setup experiment $\mathbf{Exp}^{\mathsf{set}}_{\mathcal{A}}$: In our first experiment $\mathbf{Exp}^{\mathsf{set}}_{\mathcal{A}}$, the adversary is given unlimited access to above oracles. Setup is run over a file F and the output given to the adversary, who decides where to place the bits of F.

```
Experiment $\mathbf{Exp}^{\mathsf{set}}_{\mathcal{A}}(R)$
$\mathcal{O}_{\mathsf{dev}}$.init($R$)   % Set $R^{\mathsf{in}}$ and $R^{\mathsf{out}}$.
$(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{KGen}(1^{\lambda})$
$F \leftarrow \mathcal{A}^{\mathcal{O}^*_{\mathsf{dev}}}$("setup")(pk)
$(F^*,\eta,\mathsf{pp}) \leftarrow \mathsf{St}(\mathsf{sk},\mathsf{pk},F)$
$\mathcal{A}^{\mathcal{O}^*_{\mathsf{dev}}}$("setup")$(F^*,\eta,\mathsf{pp})$
$\mathcal{O}_{\mathsf{dev}}$.clearDynamic()   % To prevent storing $F^*$ in $\mathsf{M_{dyn}}$.
Output $(\eta,\mathsf{pp},F)$
```

Challenge experiment $\mathbf{Exp}^{\mathsf{cha}}_{\mathcal{A}}$: In $\mathbf{Exp}^{\mathsf{cha}}_{\mathcal{A}}$, the adversary $\mathcal{A}$("chal") responds to a 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.

```
Experiment $\mathbf{Exp}^{\mathsf{cha}}_{\mathcal{A}}(\eta,R,\mathsf{pp})$
seed $\leftarrow_{\$} \{0,1\}^*, c \leftarrow \mathsf{Chal}(\eta,\mathsf{pp},\mathsf{seed})$
$\pi \leftarrow \mathcal{A}^{\mathcal{O}^*_{\mathsf{dev}}}$("chal").Prove$(\eta,R,c)$   % One challenge only
Output $\mathcal{O}_{\mathsf{Verify}}(\mathsf{pp},R,c,\pi)$
```

We define the success probability as follows:

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

Extraction experiment $\mathbf{Exp}^{\mathsf{ext}}_{\mathcal{A}}$: We say a PoGeoRet scheme is sound if the file F can be extracted from the static memory of devices inside R^{in}. The extraction algorithm Extract can execute any function on devices located in R^{in} using execStatic. Success is defined as the probability that the extractor reconstructs the original file F.

```
Experiment $\mathbf{Exp}^{\mathsf{ext}}_{\mathcal{A}}(\eta,R,\mathsf{pp},F)$
$F' \leftarrow \mathsf{Extract}(\eta,R,\mathsf{pp})$   where Extract can call
  $\mathcal{O}_{\mathsf{dev}}$.execStatic(id,_) only if $\mathcal{D}[\mathsf{id}].\mathsf{loc} \in R$
Output $F = F'$
```

We define the success probability as follows:

$$\mathbf{Suc}^{\mathsf{ext}}(\eta,R,\mathsf{pp},F) = \Pr\left[\mathbf{Exp}^{\mathsf{ext}}_{\mathcal{A}}(\eta,R,\mathsf{pp},F) = 1\right]. \quad (2)$$

2) Soundness definition: Our main security definition is:

Definition 1 (Soundness)**.** *A PoGeoRet scheme is ϵ-sound w.r.t a target region R^{target} achieving a region of uncertainty R^{rou}, if for all poly-time $\mathcal{A}$ and $p_1 = 1 - \mathsf{negl}\ (\lambda)$, $p_2 = 1 - \mathsf{negl}\ (\lambda)$:*

$$\Pr\left[\mathbf{Suc}^{\mathsf{ext}}(\eta,R,\mathsf{pp},F) > p_1 \;\middle|\; \begin{array}{c} R \leftarrow R^{\mathsf{rou}}, \\ (\eta,\mathsf{pp},F) \leftarrow \mathbf{Exp}^{\mathsf{set}}_{\mathcal{A}}(R), \\ \mathbf{Suc}^{\mathsf{cha}}(\eta,R,\mathsf{pp}) > \epsilon \end{array}\right] > p_2.$$

Although we defined a region to mean a circle for convenience, this definition supports arbitrarily shaped ROUs R^{rou}.

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

E. 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.

1) Bandwidth and 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.

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 and only verified 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. The extraction experiments take place after that.

We allow the adversary to modify the static memory once at the beginning of each interval (for example the adversary might store F^* inside R^{in} during one interval and move it outside R^{in} for the rest). During every interval, the adversary requests a challenge at a time of its choosing. And at the end of each interval, a snapshot of the static memory of all the devices inside R^{in} is taken. Let $S_i = \{\mathcal{D}[\mathsf{id}].\mathsf{M}_{\mathsf{sta}}\}$ where $\mathcal{D}[\mathsf{id}].\mathsf{loc} \in R^{\mathsf{in}}$ denote the snapshot for interval i. After the epoch ends, the extraction experiment is run over the snapshots $\{S_i\}_{i=1}^{I}$. The scheme is said to be *sound* only if the file can be successfully extracted from *each of the* I *snapshots*.

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.[6] 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 $\phi = \frac{|F^*|}{1000}$.

Above we used a small interval length; this is because, shorter the interval length, smaller the storage revenue, and thereby, lower the cap ϕ. Finally, our framework is generic enough to adopt more sophisticated economic models or better cost estimates.

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) [34], [41], and very costly in practice. (To avoid this problem, a number of PoRet protocols, e.g., [18], [30], 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 geolocation—such an adversary will attempt to preserve F.

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.

IV. The GoAT protocol

We now present details of our PoGeoRet scheme, GoAT. We begin by discussing GoAT-specific modeling details (Sec. IV-A). Next we provide a brief description of the Shacham-Waters (SW) PoRet scheme [40] (Sec. IV-B), followed by GoAT using SW PoRet (Sec. IV-C). For ease of presentation, we start with a simplified version that requires high-resolution anchors. Next in Sec. IV-D, we discuss how GoAT deals with low-resolution anchors. In Sec. IV-E, we present a theorem specifying the level of security GoAT provides. Finally in Sec. IV-F, we discuss how to decentralize trust among anchors.

We also consider a second PoGeoRet design that uses Merkle Tree PoRet instead of SW. This variant has the benefit of a simpler design and avoids the cryptographic

[6] 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.

hardness assumptions required for SW. It has large proof sizes, however, so we defer it to App. D.

A. 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 II.

Network model: We approximate the Earth to be a sphere. The metric space $(\mathcal{M},\mathsf{dist})$ 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 $n = S_{\mathsf{max}}/S_{\mathsf{min}}$ is the network speedup of the adversary. These parameter values need to be decided based on empirical measurements (See Sec. V). Honest providers only need to attain the speed S_{min} for a short period of time. For example, if interval length $\beta = 1$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. IV-F.

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.

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.

B. Shacham-Waters scheme

At a high level, SW uses BLS-style signatures to facilitate proof aggregation and public verifiability.

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 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. The generated signatures are $\{\sigma_i\}_{i=1}^n$ (See SW.St in Fig. 7 in the Appendix).

Each challenge 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 requires computing a linear combination of the k file blocks to compute $\boldsymbol{\mu} = \{\mu_j\}_{j=1}^s$ and aggregating the corresponding signatures to compute σ. Crucially, the proof size does not depend on the number of challenges k and depends only on the number of sectors s.

Modifications to SW: Fig. 7 in the appendix presents our (slightly) modified version of SW scheme. We add a function SW.Commit that commits the value $\boldsymbol{\mu}$. Note that $\boldsymbol{\mu}$ is computed exactly as in the proof function. As $\boldsymbol{\mu}$ is a vector of size s, we use a vector commitment scheme. The vector commitment scheme needs to be binding and homomorphic. We use Pedersen commitments (See Fig. 9).

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.

C. GoAT *with high-resolution anchors*

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

1) Protocol: As usual, our setting involves a user U that wants to store a file F with a provider situated in a location $L_{\mathsf{P}} \in R^{\mathsf{target}}$, a target region.

Setup: U runs the PoRet setup protocol (SW.St) 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 $\{F^*,\eta,\mathsf{pp}\}$ to P.

As noted before, we assume that a set of anchors $\mathcal{T}$ is predetermined; let $\mathsf{A} \in \mathcal{T}$ be one such anchor located at L_{A}. For simplicity, in this section, we assume anchors are trusted and thus that it suffices to use the single anchor A. Other protocol parameters such as the interval length β, number of intervals per epoch I are assumed to be predetermined.

Proof generation: 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 (*GeoCommit*):* The key idea is to sandwich the file access operation between successive pings to the anchor. The PoRet commitment serves as the file access operation and the anchor helps in timing this operation. Fig. 4 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 $M_1 = \{t_1,N_1\}$.

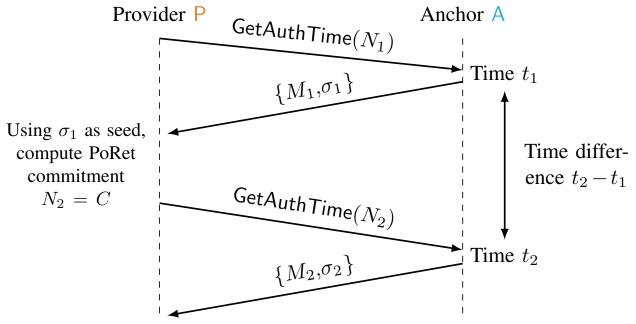

Fig. 4. The geo-commitment protocol between the provider and anchor.

Notation	Description
ϵ	Frac. of queries answered correctly
ρ	Erasure code rate
I	#intervals per epoch
β,$I\beta$	Interval length, Epoch length
$R=(L;\delta)$	Circular region defined by center L & radius δ
R^{target},R^{rou}	Target region and the ROU
n	Network speedup
S_{max},S_{min}	Network speeds (max, min)
T	Anchor transcript
Γ_{A}	Timestamp resolution of anchor A
k	Number of challenges in a PoRet
$\Delta(L_{\mathsf{A}},L_{\mathsf{P}})$	Time allocated to run GeoCommit b/w A & P.
a	Amplification factor
ϕ	Bandwidth cap
α	Grinding cap (2^{α} GetAuthTime calls)

TABLE II
NOTATION.

2) *PoRet commitment:* Use σ_1 as randomness to derive a set of challenges $\mathcal{S} \leftarrow \mathsf{SW.Chal}(\eta, \mathsf{pp}, \sigma_1)$. Now generate a commitment $C \leftarrow \mathsf{SW.Commit}(\eta,\mathcal{S})$.

3) *Ping #2:* Set nonce $N_2 = C$ and ping the anchor A again via $\mathsf{GetAuthTime}(N_2)$. Receive $T_2 = \{M_2,\sigma_2\}$ where $M_2 = \{t_2,N_2\}$.

We refer to the pair $C^{\mathsf{geo}} = \{T_1,T_2\}$ as a geo-commitment. Note that the PoRet commitment C 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 $\{C^{\mathsf{geo}}_m\}^{I}_{m=1}$.

*PoRet computation (*PoRetCompute*):* 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 exorbitant proof sizes.

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 C_I. Denote the challenge set used to compute the j^{th} PoRet commitment by $\mathcal{S}_j = \{c_{ij},v_{ij}\}^{k}_{i=1}$ where $j \in \{1,...,I\}$. The newly generated coefficients are incorporated into those for the challenge sets, $\forall j$, $\mathcal{S}^*_j = \{c_{ij},r_j v_{ij}\}^{k}_{i=1}$. The modified challenge sets are aggregated as $\mathcal{S}^* = \cup^{I}_{j=1}\mathcal{S}^*_j$.

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 $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{SW.Prove}(\eta,\mathcal{S}^*)$.

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

Proof verification: 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 $C^* = \prod^{I}_{j=1}(C_j)^{r_j}$. The proof of retrievability π^{PoRet} and C^* are verified by $\mathsf{SW.Verify}(\mathsf{pp},\mathcal{S}^*,C^*,\pi^{\mathsf{PoRet}})$.

Note that verification succeeds even with randomization of the challenge coefficients because SW.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 $t_2 - t_1 \leq \Delta(L_{\mathsf{A}},L_{\mathsf{P}})$, where Δ is a pre-agreed upon function that takes anchor, provider locations as inputs and outputs the maximum runtime of GeoCommit operations (those happening between times t_1, t_2). We now discuss how Δ is set.

2) 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 an adversarial provider.

As shown in Fig. 4, 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 $\Delta(L_{\mathsf{A}},L_{\mathsf{P}}) = t_{\mathsf{ping}} + t_{\mathsf{com}}$.

Elapsed time for the GetAuthTime API call depends on the physical distance between the anchor and provider. We have $t_{\mathsf{ping}} = \mathsf{rtt}_{\mathsf{max}}(L_{\mathsf{A}},L_{\mathsf{P}}) + t_{\mathsf{proc}}$ where the first term denotes the maximum round trip time introduced in Sec. IV-A 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. V-B for details). Therefore we have:

$$\Delta(L_{\mathsf{A}},L_{\mathsf{P}}) = (2\cdot\mathsf{dist}(L_{\mathsf{A}},L_{\mathsf{P}})/S_{\mathsf{min}}) + t_{\mathsf{start}} + t_{\mathsf{proc}} + t_{\mathsf{com}}. \tag{3}$$

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

(SW.Commit) as a means to commit to a PoRet proof before generating the proof itself.

3) Grinding attacks: Since GeoCommit protocol is prover-initiated, an adversarial prover can exploit by re-running the protocol. For example, an adversary could save on storage by only storing a portion of the file, and repeatedly querying 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 (In practice, this number needs to be set based on the actual cost to make many API calls). 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.

4) Coefficient randomization: Randomization at the end of an epoch is necessary to ensure that the PoRet commitments $\{C_i\}$ are correctly computed in every interval of the epoch. If the ratio between any two random coefficients was predictable, e.g., say $\tau = r_i/r_j$ was known for some $i \neq j$, then the adversary could skip file access in the ith interval and set C_i to random; C_j is set such that the product of commitments is same as that computed by a honest provider, $C_i^{r_i} C_j^{r_j} = H_i^{r_i} H_j^{r_j}$, so $C_j = (H_i(C_i)^{-1})^\tau H_j$. H_i and H_j are the actual ith and jth PoRet commitments—the adversary can compute them both 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 C_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].

D. GoAT *with low-resolution anchors*

The GeoCommit 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. The first and last operation are anchor pings. 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*.

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 $a = \lfloor \Gamma_{\mathsf{A}} / \Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \rfloor - 1$.[7] Below, we explain how to time proof execution in order to ensure receipt of $a+1$ transcripts with the same timestamp.

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

A general GeoCommit protocol for *any* anchor, low- or high-resolution, is specified in Fig. 6, 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 subsequence with the same timestamp (given stable network conditions). The final proof will only include the desired sub-sequence; extra transcripts can be discarded. The intuition here is that a' executions guarantees seeing two time changes (i.e., three distinct timestamps), therefore one resolution tick is fully covered, which in turn guarantees $\lfloor \Gamma_{\mathsf{A}} / \Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) \rfloor$ transcripts will have the same timestamp. As noted, this will not work if the network conditions are unstable, and other mechanisms like retries are needed in practice.

Effect on geolocation: The use of amplification has a small effect on the radius of ROU, explained through an example. Suppose $\Delta(L_{\mathsf{A}}, L_{\mathsf{P}}) = 250$ms, $\Gamma_{\mathsf{A}} = 1000$ms. Applying the above formula, we get $a = 3$, i.e., 4 pings are needed. But this leaves some "extra time"—for example, if the anchor's clock times at the moment of receipt of the 4 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.

[7] 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.

E. GoAT *security*

Say the target region is a single location, $R^{\mathsf{target}}=(L;0)$. Then the region of uncertainty achieved by GoAT is a circle centered at anchor's location with radius δ_L given by:

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

In practice, the target region might have a small diameter, $R^{\mathsf{target}}=(L;\delta')$. As long as δ' is small, we can approximate and define the region of uncertainty as $R^{\mathsf{rou}}=(\mathsf{A};\delta'')$ where $\delta''=\max_{\{L'\in R^{\mathsf{target}}\}}\delta_{L'}$.

Theorem 1. *Let* $w=\left(\rho+\frac{\phi}{|F^*|}+1-2^{\frac{-\lambda-\alpha}{k}}\right)^k$. *Assuming* $\epsilon-w$ *is positive and non-negligible and that the CDH problem is hard in bilinear groups,* GoAT *is* ϵ*-sound at a target geographic region* $R^{\mathsf{target}}=(L;\delta')$ *achieving a geolocation guarantee of* $R^{\mathsf{rou}}=(\mathsf{A};\delta'')$ *under the flexible challenge model against an economically rational adversary.*

The proof sketches are in App. C. GoAT also satisfies soundness under the non-interactive definition variants described in App. A.

F. 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 GeoCommit 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. V-B).

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.

V. Implementation and Evaluation

We implemented GoAT in approximately 2500 lines of C with support for both TLS 1.2 and Roughtime anchors. Our implementation uses TLSe [44] for TLS and Roughenough [43] for Roughtime. We use the PBC library [35] for pairings. For sensitive file operations in GeoCommit, we use the asynchronous I/O library libaio [6]. We omitted erasure coding functionality as it is not our focus.

The outline of this section is as follows. In Sec. V-A, we discuss a number of setup considerations, and in Sec. V-B, we present our evaluation results.

A. 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.

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 [31].

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 $n=S_{\mathsf{max}}/S_{\mathsf{min}}=3$ and the constant startup cost $t_{\mathsf{start}}=5$ms. These parameter choices are consistent with recent work [17] 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 security model permits a prover to make multiple proof attempts over a given interval, creating strong resilience for any network-speed fluctuations.

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 1.2, 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., [46], [31]. 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. Using faster alternatives such as ECDSA (if supported by the server) might allow somewhat better geolocation (see anchor processing times below), but many TLS servers use RSA certificates only. We find that the proof transcript length for RSA-based servers is 389 bytes,

[8] PlanetLab nodes tend to be well-connected to the internet, matching our expectation of storage provider's connection. [17] also picks geographically diverse nodes.

[9] These databases are known to have some errors [25] and a rigorous geolocation experiment like [46] would have to be done before deploying our system.

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. To get better geolocation, we open TCP connections prior to the start of the geo-commitment protocol. This eliminates the unnecessary communication time incurred by performing the first round trip during GeoCommit.

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 (the server belongs to Cloudflare, a popular CDN provider). We find that the proof transcript length for Roughtime is 360 bytes.

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

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

3) GoAT parameters: We now briefly discuss how various GoAT parameters are set. App. B contains a discussion over practical considerations related to parameterization.

For SW PoRet, we use an asymmetric pairing-friendly curve "g149" [35]. The rationale being that it took the least time to perform a vector commitment, i.e., a fixed-base multi-exponentiation operation, among the options provided in the library.[10] Fast multi-exponentiation helps both in finer geolocation and faster verification. Except in one experiment below, we set the number of sectors per block, $s = 96$.

As we discuss in Sec. IV-E, $(\rho + \frac{\phi}{|F^*|} + (1 - 2^{-(\lambda+\alpha)/k}))^k$ needs to be negligible. Assuming the grinding constraint $\alpha = 40$, one set of parameters to achieve 80-bit security are code rate $\rho = 0.33$ and number of challenges $k = 170$;[11] note that the bandwidth cap is set to $\phi = 0.001|F^*|$ using the economic analysis from Sec. III-E1.

For the experiments below, we set the number of challenges to just $k = 100$; we expect minimal impact on our results.

Remaining parameters: In eq. (3), 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 $t_{\mathsf{cproc}} = 1.5$ms based on code benchmarks. The expected commitment compute time t_{com} is also fixed based on benchmark results discussed below.

B. Evaluation

We evaluate GoAT through several benchmarks and perform a real-world experiment over a week (Sec. V-B1). 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. V-B1, 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. IV, PoRet commit time has a direct impact on the ROU radius. Table III 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 fast SSD seeks. This is a key reason why GoAT achieves good geolocation accuracy. If instead a naïve approach of computing a full PoRet was taken, the geolocation accuracy would have been 75x worse! (This is based on numbers from Table IV and Table III.)

The times in Table III are largely constant except for an abrupt jump at 64GB. This happens because the cache is no longer useful. Direct I/O[12] is used to read from bigger files (last 2 rows) to improve seek times. The expected time to compute a commitment is set to $t_{\mathsf{com}} = 2$ms for later experiments.

Computation costs: Table IV presents the time taken for the PoRetCompute 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 1000 intervals, PoRetCompute takes about 21s and Verify takes around 19s. 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 PoRetCompute 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: Setting parameters from above, GoAT proof size is $1844 + |T|I(a+1)$ bytes. The first half of the equation (constant part) is contributed by the PoRet proofs, while the second half by anchor transcripts.

[10] We did not specifically tune curve parameters nor did we use hardware optimizations. Both are possible opportunities to optimize.

[11] Another set of parameters with higher code rate is $\rho = 0.5$, $k = 240$.

[12] 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.

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 III
TIME TAKEN FOR PORET COMMIT. STANDARD DEVIATIONS IN BRACKETS.

#intervals	PoRetCompute (ms)	Verify (ms)
1	76.67 (0.40)	55.48 (0.55)
10	275.70 (0.81)	234.28 (5.59)
100	2,175.16 (10.46)	1,992.32 (61.03)
1000	21,227.36 (102.99)	19,404.71 (411.75)

TABLE IV
COMPUTATION TIME OF PoRetCompute AND Verify (VS) NUMBER OF INTERVALS PER EPOCH. STANDARD DEVIATIONS IN BRACKETS.

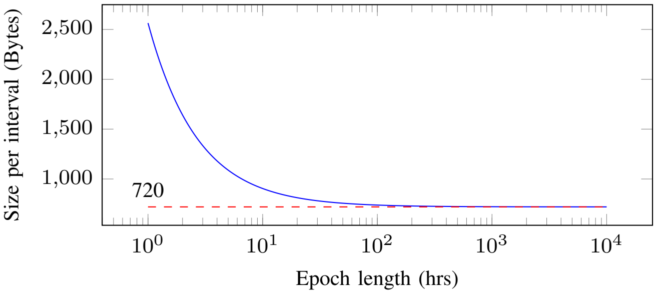

Fig. 5. For a fixed interval length of 1hr, the proof size per interval when using a Roughtime anchor plotted against the epoch length. Red line (720B) corresponds to the size of two Roughtime transcripts.

Concretely, assuming Roughtime anchors ($a = 1$) and $I = 100$, GoAT proof size is 36.96KB with a dominating 35.15KB of anchor transcripts. With a TLS anchor requiring amplification factor $a = 20$, the proof grows to 740.08KB.

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. 5. The plot converges at 720B, the size of 2 Roughtime transcripts.

GoAT is extremely communication-efficient compared to alternative PoGeoRet designs largely due to the use of Shacham-Waters. For example, with use of a Merkle-Tree PoRet scheme, the proof size per interval would remain constant at around 100KB, which is in fact too high to show in the plot (138x larger than GoAT). The PoRet-compression optimizations also help, but to a lesser extent: without them, the per-interval proof size would be 2564B, 3.5x bigger than GoAT.

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 GeoCommit protocol alone as it is the sole operation affected by network conditions. Prior works [28] have observed network stability over long time periods, and conclude that network instability is frequent but most often transient. So we handle failures in GeoCommit by simply retrying until success. Concretely, the number of retries is capped at 30 with a gap of 1 second between retries. In this process, we count the number of retries needed to succeed and the false rejection rate, if any. Under ideal network conditions, 0 retries are expected.

We run the GoAT 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 GeoCommit protocol is run for 10 days at NV (525 intervals) and 7 days at LON (347 intervals).

Table V 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.

The ROU radius to distance ratio δ/d is useful to understand the key factors contributing to the quality of geolocation. Recall from Sec. IV that this ratio is given by $(n+((t_{\mathsf{com}}+t_{\mathsf{start}}+t_{\mathsf{proc}})S_{\mathsf{max}}/2\mathsf{dist}(L_{\mathsf{A}},L_{\mathsf{P}})))$. For providers farther away from the anchor, we see the ratio converging to speedup $n=3$ 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. B 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".

The last column in Table V 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").

VI. RELATED WORK

A long line of works aim to prove correct file storage by a storage provider, e.g., Proof of Retrievability [40], [30], Proof of Data Possession [12], [26] and more recently

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 V

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.

Proof of Replication [23], [19], [24], [14]. To the best of our knowledge, only few works [16], [47] aim to prove file location, but they operate in a trusted-verifier setting. Our work is the first to consider decentralized trust among verifiers.

While GoAT can be used to construct a PoRep, we do not explicity compare GoAT's performance with a PoRep as existing PoReps [23] are typically designed to be mining functions unlike a simple PoGeoRet. Extending GoAT to support mining is a direction of future work (See App. B).

Most geolocation technologies in use today (e.g., GPS, Bluetooth beacons [29], [47]) rely on trusted verifiers and are hence unusable in decentralized systems. FOAM [21] generates a proof-of-location using permissionless anchors that perform triangulation in small zones, but their approach suffers from bootstrapping problems.

Many prior works [31], [46], [16] have used distance-bounding like techniques for geolocation. Our protocol is also a novel application of the same underlying technique: typically distance bounding works by verifier sending a challenge and measuring the time taken for a prover to respond, whereas we leverage public timestamping servers to track time instead of a verifier. Our approach has some similarities to [46], which uses existing anchors with known locations (educational / govt servers) to geolocate IP addresses.

VII. 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.

VIII. ACKNOWLEDGMENTS

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

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

REFERENCES

[1] Alexa top sites. https://www.alexa.com/topsites, 2021. [Accessed Apr 2021].

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

[3] Cryptocurrency prices by market cap, 2021. [Accessed May 2021].

[4] Drand - distributed randomness beacon, 2021. [Accessed May 2021].

[5] Filecoin aims to use blockchain to make decentralized storage resilient and hard to censor, 2021. [Accessed May 2021].

[6] Linux-native asynchronous i/o access library. https://pagure.io/libaio, 2021. [Accessed Apr 2021].

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

[8] Ssd userbenchmarks - 1058 solid state drives compared. https://ssd.userbenchmark.com/, 2021. [Accessed Apr 2021].

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

[10] Amazon. Aws ec2 costs. https://aws.amazon.com/ec2/pricing/on-demand/, 2021. [Accessed Apr 2021].

[11] Amazon. Aws s3. https://aws.amazon.com/s3/, 2021. [Accessed Apr 2021].

[12] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM CCS*, pages 598–609, 2007.

[13] J. Benet. IPFS - content addressed, versioned, P2P file system. *CoRR*, abs/1407.3561, 2014.

[14] J. Benet, D. Dalrymple, and N. Greco. Proof of replication. *Protocol Labs, July*, 27:20, 2017.

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

[16] K. Benson, R. Dowsley, and H. Shacham. Do you know where your cloud files are? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 73–82, 2011.

[17] I. N. Bozkurt, A. Aguirre, B. Chandrasekaran, P. B. Godfrey, G. Laughlin, B. Maggs, and A. Singla. Why is the internet so slow?! In *International Conference on Passive and Active Network Measurement*, pages 173–187. Springer, 2017.

[18] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. *Journal of Cryptology*, 30(1):22–57, 2017.

[19] E. Cecchetti, B. Fisch, I. Miers, and A. Juels. Pies: Public incompressible encodings for decentralized storage. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1351–1367, 2019.

[20] M. Clark. Nfts, explained, Mar 11, 2021. [Accessed Apr 2021].

[21] F. Corp. Foam: The consensus driven map of the world, 2018. [Accessed May 2021].

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

[23] B. Fisch. Poreps: Proofs of space on useful data. *IACR Cryptol. ePrint Arch.*, 2018:678, 2018.

[24] B. Fisch, J. Bonneau, N. Greco, and J. Benet. Scaling proof-of-replication for filecoin mining. *Benet//Technical report, Stanford University*, 2018.

[25] P. Gill, Y. Ganjali, B. Wong, and D. Lie. Dude, where's that ip? circumventing measurement-based ip geolocation. In *Proceedings of the 19th USENIX conference on Security*, pages 16–16, 2010.

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

[27] E. L. Harding, L. J. Acevedo, and L. R. Dailey. Data localization and data transfer restrictions. https://www.natlawreview.com/article/data-localization-and-data-transfer-restrictions/, August 2021. [Accessed Aug 2021].

[28] T. Høiland-Jørgensen, B. Ahlgren, P. Hurtig, and A. Brunstrom. Measuring latency variation in the internet. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 473–480, 2016.
[29] K. E. Jeon, J. She, P. Soonsawad, and P. C. Ng. Ble beacons for internet of things applications: Survey, challenges, and opportunities. *IEEE Internet of Things Journal*, 2018.
[30] A. Juels and B. S. Kaliski Jr. Pors: Proofs of retrievability for large files. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 584–597, 2007.
[31] E. Katz-Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe. Towards ip geolocation using delay and topology measurements. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 71–84, 2006.
[32] P. Labs. Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf, July 19, 2017. [Accessed Apr 2021].
[33] S. Labs. Storj: A decentralized cloud storage network framework. https://www.storj.io/storjv3.pdf, October 30, 2018. [Accessed Apr 2021].
[34] S.-J. Lin, T. Y. Al-Naffouri, Y. S. Han, and W.-H. Chung. Novel polynomial basis with fast fourier transform and its application to reed–solomon erasure codes. *IEEE Transactions on Information Theory*, 62(11):6284–6299, 2016.
[35] B. Lynn. The pairing-based cryptography library. https://crypto.stanford.edu/pbc/, 2021. [Accessed Apr 2021].
[36] C. Mellor. Ssds will crush hard drives in the enterprise, bearing down the full weight of wright's law. https://blocksandfiles.com/2021/01/25/wikibon-ssds-vs-hard-drives-wrights-law/, January 25, 2021. [Accessed Apr 2021].
[37] C. Patton. Roughtime: Securing time with digital signatures. https://blog.cloudflare.com/roughtime/, 2018. [Accessed Apr 2021].
[38] Qualys. Ssl pulse. https://www.ssllabs.com/ssl-pulse/, April 11, 2021. [Accessed Apr 2021].
[39] P. Reinheimer and W. Roberts. Global ping statistics → manhattan. https://wondernetwork.com/pings/Manhattan. [Accessed Apr 2021].
[40] H. Shacham and B. Waters. Compact proofs of retrievability. In *International conference on the theory and application of cryptology and information security*, pages 90–107. Springer, 2008.
[41] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 325–336, 2013.
[42] A. Shokrollahi. Raptor codes. *IEEE transactions on information theory*, 52(6):2551–2567, 2006.
[43] S. Stock. Roughenough. https://github.com/int08h/roughenough, 2021. [Accessed Apr 2021].
[44] E. Suica. Single c file tls 1.2/1.3 implementation. https://github.com/eduardsui/tlse/, 2021. [Accessed Apr 2021].
[45] D. Vorick and L. Champine. Sia: Simple decentralized storage. https://sia.tech/sia.pdf, November 29, 2014. [Accessed Apr 2021].
[46] Y. Wang, D. Burgener, M. Flores, A. Kuzmanovic, and C. Huang. Towards street-level client-independent ip geolocation. In *NSDI*, volume 11, pages 27–27, 2011.
[47] G. J. Watson, R. Safavi-Naini, M. Alimomeni, M. E. Locasto, and S. Narayan. Lost: location based storage. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pages 59–70, 2012.

APPENDIX A
NON-INTERACTIVE PROOFS OF GEOGRAPHIC 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 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.

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 $\mathsf{true\_time}$, 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.

#sectors	Time (ms)
64	1.16 (0.01)
128	2.08 (0.01)
256	4.16 (0.02)

TABLE VI
TIME TAKEN FOR SW PORET COMMIT WITHOUT THE FILE READ OPERATION AS A FUNCTION OF THE NUMBER OF SECTORS.

A. *Security properties*

Non-interactive Completeness: The completeness definition is the same as before, except that no challenges are issued by the verifier.

Non-interactive Soundness: The changes to the security experiments are as follows. 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. We define

$$\mathbf{Suc}^{\mathsf{cha}'}(\eta,R,\mathsf{pp}) = \Pr\left[\mathsf{NIVerify}(\mathsf{pp},R)=1\right]. \tag{4}$$

The extraction experiment is same as before. The new soundness definition is also roughly the same as before, barring the change to challenge experiment:

Definition 2 (NISoundness)**.** *A NIPoGeoRet scheme is ϵ-sound w.r.t a target region R^{target} achieving a region of uncertainty R^{rou}, if for all poly-time $\mathcal{A}$ and $p_1 = 1-\mathsf{negl}\ (\lambda)$, $p_2 = 1-\mathsf{negl}\ (\lambda)$:*

$$\Pr\left[\mathbf{Suc}^{\mathsf{ext}}(\eta,R,\mathsf{pp},F) > p_1 \;\middle|\; \begin{array}{c} R \leftarrow R^{\mathsf{rou}}, \\ (\eta,\mathsf{pp},F) \leftarrow \mathbf{Exp}^{\mathsf{set}}_{\mathcal{A}}(R), \\ \mathbf{Suc}^{\mathsf{cha}'}(\eta,R,\mathsf{pp}) > \epsilon \end{array}\right] > p_2.$$

B. *Changes to* GoAT

The GoAT protocol is already mostly non-interactive. For example, recall that in GoAT, the newcomer verifies the proof by using the list of public keys of the anchors.

In the specified protocol in Fig. 10, the function Chal can be removed as a randomly chosen initial challenge is unnecessary for GoAT's security.

Appendix B
Practical considerations and Future work

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 (See Table VII). Note that higher s leads to increased PoRet commit times and thereby worse geolocation (eq. (3)).

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. V-A3, 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.

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.

3) Finding anchors: In Sec. V, 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.

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.

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.

Appendix C
Security Proofs

Recall that the PoGeoRet proof consists of I geo-commitments and a PoRet proof. Each geo-commitment 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 PoGeoRet proof.

As GoAT is a non-interactive protocol, i.e., a NIPoGeoRet scheme, we provide the proof for the non-interactive variant of the soundness definition (Def. 2).

We prove soundness of GoAT in four steps.

1) Prove that the N PoRet commitments and the PoRet proof are correctly computed, i.e., the PoRet verification protocol (PoRet.Verify) part of Verify must detect otherwise.
2) Timing-based argument to prove that the challenged file blocks used to compute a valid PoRet commitment must be in the target region, i.e., the verification protocol (Verify) must detect otherwise.
3) Prove that the extraction algorithm Extract can efficiently reconstruct ρ fraction of file blocks from 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. Note that we only provide proof sketches for the rest.

A. Part-two proof

For this part, we assume that the PoRet commitments, proofs are correctly computed and prove that the challenged file blocks must be in the desired target region during every interval.

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.

We begin with the high resolution anchors setting.

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

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

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

The time taken to receive and respond from $\mathsf{D}_{\mathsf{out}}$ during the geo-commitment protocol with A is given by $z = 2\mathsf{dist}(L_1, L_2)/S_{\mathsf{max}}$. This is because in Fig. 6 we derive challenges from anchor signatures, i.e., they arise at L_1 and must reach L_2. We can assume that the adversary probability of guessing these challenges is negligible (requires breaking selective unforgeability of the signature scheme used by the anchor which happens with negligible probability).

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

For the geo-commitment verification to succeed, it must be that $z \le \Delta(L_p, L_1)$. (See last step in Fig. 4 when $a=1$.)

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

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

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

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

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

Note on technique: One subtlety to note is that the following alternate amplification method that computes PoRet commitment only once does not work: $ping_1$, com_1, $ping_2$, $ping_3$, ..., $ping_a$, $ping_{a+1}$. At first sight it might seem like a reasonable approach as it can also fill up a large amount of time.

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

B. Remaining proofs

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

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 C_N (in PoRetCompute-SW). These coefficients are used during verification (in Verify-SW) to compute C as follows, $C = \prod_{j=1}^{N} (C_j)^{r_N}$. 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.

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.

A key difference now is that the extraction algorithm runs directly on the static memory snapshots $\{S_i\}_{i=1}^{I}$. We prove that ρ fraction of the encoded file blocks can be extracted from one of the snapshots, say S_1. Proof for the rest is similar.

Scheme	GoAT-MT	GoAT-SW
Parameters	$n=\frac{1}{b\rho}\lvert F\rvert$	$n=\frac{1}{s\rho\lambda}\lvert F\rvert$
Storage overhead	$32n+(\frac{1}{\rho}-1)\lvert F\rvert$	$n\lambda+(\frac{1}{\rho}-1)\lvert F\rvert$
Proof size	$Iak(b+32\log_2 n)+$ $I(a+1)\lvert T\rvert$	$(s+1)\lambda+$ $I(a+1)\lvert T\rvert$
PoRet.Commit time	$O(kb)$	$O(ks)$

TABLE VII
STORAGE OVERHEAD, PROOF SIZES AND COMMIT TIMES FOR GoAT-MT AND GoAT-SW.

The key question is how the new bandwidth constraint ϕ impacts 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 in the snapshot S_1 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 in S_1.

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 $\rho n \leq n$ type-1 queries to succeed, which happens in $O(n/(\epsilon-w))$ time.

The maximum number of blocks unavailable in the static memory 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 $\rho n+\gamma \leq n$, extraction happens in $O(n/(\epsilon-w))$ time, i.e., same order as before. One constraint we get is $\frac{\phi}{|F^*|} \leq g-\rho$.

We want $\epsilon-w$ to be positive and non-negligible. Therefore w needs to be negligibly small. Meaning $(\rho+\gamma/n)^k$ (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. III-E2). 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.

Geo-commitment generation (GeoCommit) **between** P **and** A

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

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

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

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

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

Geo-commitment verification by V

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

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

Fig. 6. Geo-commitment protocols.

APPENDIX D
GoAT-MT: GoAT WITH MERKLE TREES

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

A. *Merkle Tree PoRet*

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

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

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

The new commit function, MT.Commit, involves computing a hash over the requested file blocks; note that the Merkle tree is not used in this computation. Figure 11 presents the scheme.

B. GoAT-MT *protocol*

Figure 10 also specifies GoAT-MT, denoted by "MT" in the figure. The GeoCommit protocol is same as before. In PoRetCompute, unlike GoAT-SW, all PoRets are computed individually thereby leading to high proof sizes. The proofs are later verified in Verify.

Table VII presents the concrete storage overhead, proof size and the algorithmic complexity of commit for GoAT-MT

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.

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

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

Fig. 7. The Shacham-Waters PoRet scheme with an extra commitment step.

Proof of Retrievability

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

Fig. 8. Publicly verifiable PoRet API. PoRet.Commit is the only addition compared to prior modeling [30].

and GoAT-SW. We note that the proof sizes shown in the table does not account for several practical optimizations one might use to compress GoAT-MT proofs, e.g., remove some portion of the top half of the tree as it is likely the same. We expect these optimizations to bring down proof sizes by a small constant only.

Pedersen commitment

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

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

Fig. 9. Pedersen vector commitment scheme

PoGeoRet 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 $(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{PoRet.KGen}(1^\lambda)$.
- $(F^*, \eta, \mathsf{pp}) \leftarrow \mathsf{St}(\mathsf{sk}, \mathsf{pk}, F)$: U runs $(F^*, \eta, \mathsf{pp}) \leftarrow \mathsf{PoRet.St}(\mathsf{sk}, \mathsf{pk}, F)$ and picks a geographic region $R = (L; \delta_L)$. Values $\{F^*,\mathsf{pp},R\}$ are given to P situated at L.
- $c \leftarrow \mathsf{Chal}(\eta,\mathsf{pp},\mathsf{seed})$: Derive 32-byte values $c = \{\mathsf{seed}_m\}_{m=1}^I \leftarrow \mathsf{PRF}(\mathsf{seed})$.
- $\pi^{\mathsf{geo}} \leftarrow \mathsf{Prove}(\eta, R, c, \mathsf{pp})$: P selects an anchor $\mathsf{A} \in \mathcal{T}$ based on the input region R. P generates geo-commitments via GeoCommit once every interval, and proofs of retrievability via PoRetCompute once every epoch. Send $\pi^{\mathsf{geo}} = \left\{ \{C_m^{\mathsf{geo}}\}_{m=1}^I, \pi^{\mathsf{PoRet}} \right\}$ to V.
  - $C^{\mathsf{geo}} \leftarrow \mathsf{GeoCommit}(\eta, \mathsf{seed}_m, \mathsf{pp})$: P runs the protocol in Fig. 6 with the anchor A.
  - $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{PoRetCompute}(\eta, \{C^{\mathsf{geo}}\}_{j=1}^{Ia})$: Let $N = Ia$. Let $\mathcal{S}_j$, C_j be the PoRet challenge set and the commitment generated to compute C_j^{geo}. The proof computation process differs by the PoRet scheme:
    - ∗ **MT**: Run $\forall j$, $\pi_j^{\mathsf{PoRet}} \leftarrow \mathsf{MT.Prove}(\eta, \mathcal{S}_j)$. $\pi^{\mathsf{PoRet}} = \left\{\pi_j^{\mathsf{PoRet}}\right\}_{j=1}^N$.
    - ∗ **SW**: P derives N random coefficients in $\mathbb{Z}_p$ from the last PoRet commitment, $\{r_j\}_{j=1}^N \leftarrow \mathsf{PRF}(C_N)$. 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 $\pi^{\mathsf{PoRet}} \leftarrow \mathsf{SW.Prove}(\eta,\mathcal{S}^*)$.
- `Fpor/Fother/Succ` $\leftarrow \mathsf{Verify}(\mathsf{pp}, R, c, \pi^{\mathsf{geo}})$: V unpacks $c = \{\mathsf{seed}_m\}$, $\pi^{\mathsf{geo}} = \left\{ \{C_m^{\mathsf{geo}}\}_{m=1}^I, \pi^{\mathsf{PoRet}} \right\}$. First π^{PoRet} is verified using below specified method. Geo-commitments verification is in Fig. 6: if it fails, `Fother` is returned.
  - **MT**: Denote $\pi^{\mathsf{PoRet}} = \left\{\pi_j^{\mathsf{PoRet}}\right\}_{j=1}^N$. Check $\forall j, \mathsf{MT.Verify}(\mathsf{pp},c_j,C_j,\pi_j^{\mathsf{PoRet}}) = 1$. Else return `Fpor`.
  - **SW**: 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 $C = \prod_{j=1}^N (C_j)^{r_j}$ and checks if $\mathsf{SW.Verify}(\mathsf{pp},\mathcal{S}^*,C,\pi^{\mathsf{PoRet}}) = 1$. Else return `Fpor`.

Fig. 10. The GoAT proof of geo-retrievability schemes. It includes both the Shacham-Waters (**SW**) and the Merkle-Tree (**MT**) variants.

Merkle tree PoRet scheme

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

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

Fig. 11. The Merkle tree PoRet scheme with an extra commitment step.