Accountable Proof of Ownership for Data using Timing Element in Cloud Services

Mainul Mizan, Md Lutfor Rahman, Rasib Khan, Munirul Haque, Ragib Hasan
SECRETLab, Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama, USA
Email: {mainul, lrahman, rasib, mhaque, ragib}@cis.uab.edu

***Abstract*—While many cloud storage and infrastructure systems exist today, none of them provide a mechanism for accountability of stored or user generated content. This lack of security support has been a major hurdle for auditing documents, claiming data possession, and proof of authorship. In this paper, we present a novel idea for secure accountability of timing element for data in massively scalable systems. The proposed scheme allows a service provider to incorporate timing accountability of data generated at the provider, by requesting proofs from accountability servers in the cloud. Additionally, the size of the proof is independent of the data size and is a unique feature of our system design. The scalability of the system have been evaluated using the Amazon EC2.**

***Keywords*-accountability, asserted proofs, cloud services, data ownership, timing element, trust score**

I. Introduction

Cloud computing has become one of the most popular computing paradigms in recent years for its flexibility, reliability, hassle free maintenance, ease of use, and low cost. Cloud computing opens a new horizon of data storage. According to Gartner Inc. [6], more than 50 percent of top 1000 companies of the world will have stored customer-sensitive data in a public cloud by the year 2016.

An increasing amount of customer sensitive data stored in cloud has made accountability preservation of data a crucial requirement for cloud based services. We refer to accountability as the ownership of data with respect to time. In this context, an accountable system thus ensures an undeniable, certifiable, and tamper evident mechanism for generating and verifying proofs of ownership of data [19]. Thus, preservation of history is necessary on numerous grounds, such as, auditing, financial recording, claiming data ownership, and preserving medical records. It can be especially useful for patents, proving authorship and other intellectual property litigation.

A system with accountability protects its actors from false accusations. To illustrate the importance of accountability, we present the following scenario.

John felt severe pain on his chest and decided to visit Dr. Alice for his treatment. Dr. Alice referred John to Dr. Eve, who performed the test and sent the test results to Dr. Bob. Later, John suffers a severe health problem because of a misdiagnosis by Dr. Eve. Based on that, John sues Dr. Alice and Dr. Bob for malpractice. At this point, Dr. Alice and Dr. Bob want to show the medical record history (John ⇒ Alice ⇒ Eve ⇒ Bob) of John and prove their innocence. Conversely, Dr. Eve has an intention to hide her actions by retrospectively changing the history in John's medical records by colluding with John. Using our scheme in this hypothetical scenario, John's medical report from Dr. Eve will have accountable and tamper evident proof. Thus, verification of the proof will help Dr. Alice and Dr. Bob establish their innocence for the accusation.

In this paper, we propose a novel approach for accountability preservation of data ownership utilizing the timing element. Our scheme ensures an unforgeable and verifiable proof mechanism for data generated by application service providers. The contributions in this paper are as follows:

- We introduce a novel scheme for accountability preservation of data ownership for cloud-based service providers. To the best of our knowledge, this is the first accountability preservation approach applied to cloud data through the novel incorporation of timing element.
- We successfully verify that the proposed system is capable of withstanding various types of attack scenarios. We present an exhaustive security analysis to ensure mitigation of all possible attacks on the proposed architecture..
- Our proposed scheme is massively scalable for both application service providers and accountability service providers. Additionally, our system performs equally well for large and small file sizes, since the proof size remains the same irrespective of the sizes of the original data. We implemented and deployed a real-life prototype on Amazon Elastic Compute Cloud (EC2), and the results have been presented to support our claim.

The remainder of this paper is organized as follows. We discuss the system models in section II. The detailed description of our accountability preservation protocol is presented in section III. Security analysis on different attacks has been shown in section IV, followed by evaluation of results in section V. Related works and some limitations of our proposed system are discussed in section VI and VII respectively. Finally, the conclusion and plans for future work are included in section VIII.

II. System Model

In this section, we present the entities of the system, their functionalities, and the possible attacks on the accountability preservation architecture.

A. System Components

We define the following entities for our proposed scheme. The data owner is an application service provider *AP*, who has his own set of customers. Secondly, there is a group of accountability servers *AS*, who combinedly provide the service for accountability preservation. Finally, there are auditors who are able to verify claims of data ownership with a given proof.

B. Functional Description

An application service provider manipulates and possesses data for his customers. Such providers may include a web application, and/or a cloud storage servers. The accountability servers are asserted proof providers for a given data. An application provider performs a mutual authentication with one of the accountability servers, and sends the data to request for accountability preservation. The accountability server then creates a proof for the data, and communicates with a subset of other accountability servers for assertions on the proof. The asserted proof is then sent back to the application service provider as an accountable proof of ownership of the data.

For verification, the application provider presents a data and the corresponding asserted proof to an auditor. The auditor looks up the assertions in the proof, and requests each of the accountability servers for an audit proof. The auditor verifies the claim of data possession based on a consistent timing pattern over the asserted proof in comparison to the audit proof. Upon successful verification, the auditor presents a validated timing for the given data.

C. Threat Model

The possible attacks on an accountable system can be summarized as follows:

- **False or Cloned Proof:** In this attack, a malicious party produces a false proof of ownership for a given data. Another form of this attack can be forging a genuine proof to represent a different data than the proof was initially created for.
- **Time Tampering:** A malicious accountability server can provide wrong timing information when requesting for proof of ownership. Conversely, a malicious party can alter timing information of accountability element on the proofs when requesting for audit proofs.
- **Mutual Collusion:** Multiple accountability servers can collude with an application provider and produce fake proofs with false timing information and/or false data.
- **Confidentiality and Privacy:** Confidential and sensitive information about customer data, such as, usage patterns and data size, can be leaked by an external adversary or a compromised accountability server.
- **Auditor Falsification:** A compromised auditor can collude with an accountability server and produce fake proofs to frame an honest client. He can also give arbitrarily false decision to deny a valid timing for a set of proofs.

III. An Accountability Preserving Protocol

In this section, we present the secured architecture and schematic description of a accountable preserving protocol. We consider an application service provider *(AP)*, being the data owner, requests and gets an accountable proof of possession from an Accountability server *(AS)*. Additionally, our protocol includes specification of a '*trust score*' and a '*proof configuration*', the description of which have been included in the following sections.

A. Trust Score

We specify a trust score for each *AS*. To assign the trust score for each *AS*, we define two parameters: *n* and *p*. Here, n is the total number of *AS*s required to generate a proof, and p is the number of malicious/colluding *AS*s. A trust score δ is defined as $\delta = \frac{1}{n-p}$, where $p = \lfloor \frac{n-1}{2} \rfloor$, and δ is equally distributed among all *AS*s. We define *p* as above, such that our scheme is tamper evident for $2p < n$. The explanation is provided later in the security analysis in section IV-B.

B. Proof Configuration

The proof configuration is a pre-determined contract between the auditor and the application service provider. The configuration defines the requirements of any proof presented to the auditor by the data owner to claim ownership of a data item. We define proof configuration as a specification for each accountability preserving proof, which includes *n*, a trust threshold ξ, and an optional list of [*Preferred_AS* : *Elevated_Trust_Score*] value pairs. A proof configuration is thus denoted as $(n, \xi, [list])$. Here, an elevated trust score is a trust value assigned to an *AS* by an auditor according to the auditor's preference. The trust threshold ξ is a value specified by the auditor to enforce a minimum acceptable limit for the validity of the proof.

A notable thing is that an *AS* can have different configurations with different trust scores. Additionally, it is expected that the elevated trust value of the listed *AS*s will be higher than it would be otherwise on an even distribution.

The elevated trust scores $\delta_1, \delta_2, ..., \delta_m$ for $AS_1, AS_2, ..., AS_m$ are included in the proof configuration for the preferred list of *m* number of *AS*s. The sum of the elevated trust scores $S_{E\delta}$ for all *AS*s from the preferred list can be defined as:

$$S_{E\delta} = \sum_{1}^{m} \delta_i$$

Therefore, the evenly distributed trust score δ for *AS*s outside the preferred list can be rewritten as:

$$\delta = \frac{1 - S_{E\delta}}{n - p}$$

Note, that if the given list of preferred *AS*s is empty, equally distributed trust scores for all *AS*s is the same as before, where $\delta = \frac{1}{n-p}$.

C. Mutual Authentication

All communication sessions initiate after an application service provider and an accountability server are mutually authenticated in a three round protocol. The phases of the authentication protocol are presented as follows.

The application service provider *(AP)* sends an authentication request to the Accountability server *(AS)* as shown below:

$$Auth_{req} = (E_{PubAS}(AS_{id}, NC_{AP}), AP_{id}) \quad (1)$$

In expression 1, the *AP* sends an authentication request $Auth_{req}$ with a nonce NC_{AP} and identity AS_{id} of the *AS*, encrypted with the public key E_{PubAS} of *AS*, and concatenates its own id AP_{id} with it.

Upon receipt of the authentication request, the *AS* responds to *AP* as follows:

$$Auth_{rep} = (AP_{id}, E_{PubAP}(AP_{id}, NC_{AP}, NC_{AS})) \quad (2)$$

Here, the *AS* responds with an authentication reply $Auth_{rep}$ (expression 2), with the identity AP_{id} of *AP*, along with the encrypted value of *AP*'s identity, the nonce NC_{AP} received previously, and a new nonce NC_{AS}, using the public key E_{PubAP} of *AP*.

Finally, the *AP* replies to *AS* with the following message:

$$Auth_{fin} = (E_{PubAS}(AS_{id}, NC_{AS})) \quad (3)$$

The *AP* responds with the $Auth_{fin}$ message (expression 3), created using *AS*'s identity AS_{id} and the nonce NC_{AS} received from *AS*, encrypted using the public key E_{PubAS} of *AS*. The mutual authentication has thus completed, and the AP enters the asserted accountable proof collection phase.

D. Asserted Accountable Proof Collection

The *AP* generates and maintains its data. Upon requirement, the *AP* requests for accountable proof from one of the *AS*s, and preserves the attested accountable proofs. An *AS* preserves the proof requests received from the *AP*s, as well as assertion requests from other *AS*s. The sequence diagram for the protocol with a proof configuration of three servers is illustrated in Figure 1.

In our scheme, we identify each proof with using *GUID*, a globally unique identifier during the lifetime of an *AS*. The protocol requires no precise global clock, and decentralized time management from individual *AS*s and *AP*s suffice the requirement for the functionality of the scheme.

Initially, we require the *AP* to mutually authenticate with one of the *AS*s. Once authenticated, a shared key is set up between the *AP* and the *AS* using Diffie-Hellman *(D-H)* key exchange protocol [7]. This key is used for all further communication between the two parties.

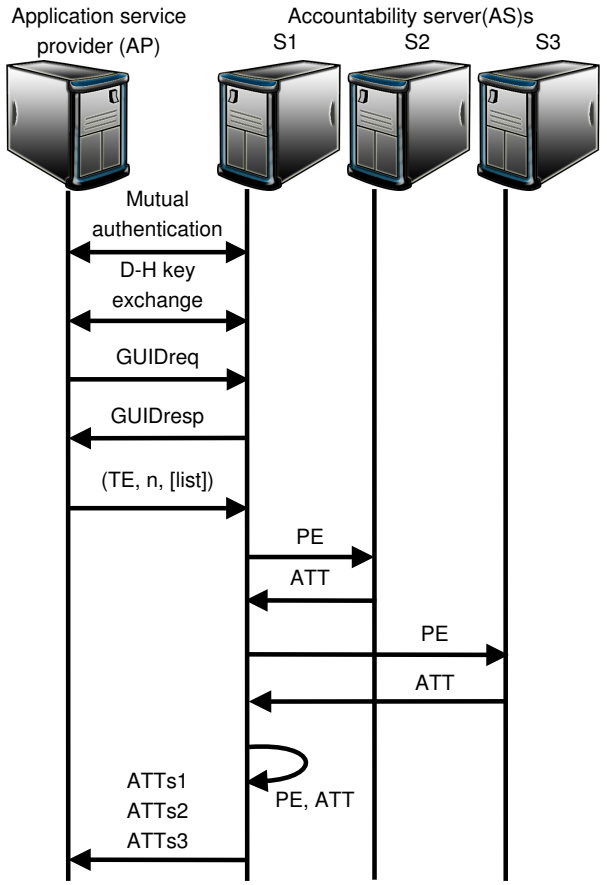

Fig. 1. The sequence diagram presenting communications between an application service provider(AP), and accountability servers(AS) of an asserted accountable proof collection for a proof configuration of 3 servers.

To begin the proof collection protocol, the *AP* requests a *GUID* from the *AS*. The *AS* replies with a *GUID* and the *AP*'s id, encrypted with the private key E_{PvtAS} of the *AS*.

$$GUID_{resp} = E_{PvtAS}(GUID, AP_{id}) \quad (4)$$

After the $GUID_{resp}$ is received, the *AP* creates a timing element *TE*. The *TE* includes the *Data*, the received *GUID*, the *AP*'s id, and a timestamp TS_{AP}, encrypted with *AP*'s private key.

$$TE = E_{PvtAP}(Data, GUID, AP_{id}, TS_{AP}) \quad (5)$$

At this point, we assume that the *AP* has already obtained a proof configuration from an auditor with the pre-determined agreements. The *AP* then includes the number of required *AS*s and the list of preferred *AS*s from the proof configuration, and sends the information to the communicating *AS*.

$$(TE, n, [list]) \quad (6)$$

Once the *AS* receives the information from the *AP*, the *AS* creates a proof element *PE*, and sends it to the other *AS*s for assertion. The other asserting *AS*s are selected based on the list of *m* preferred *AS*s, and (*n-m*) randomly selected *AS*s.

$$PE = E_{PvtAS}(HTE, GUID, AS_{id}) \tag{7}$$

Here, the *PE* includes the SHA-256 hash *HTE* of the timing element, the *GUID*, and *AS*'s id, encrypted with the given *AS*'s private key.

Each requested server, $AS_1, AS_2, ..., AS_n$, verifies the GUID is unique for the requesting *AS*. If successfully verified, the asserting AS_x saves the *PE*, and responds with a corresponding attestation ATT_{ASx} for the given request.

$$ATT_{ASx} = E_{PvtASx}(PE, ASx_{id}, TS_{ASx}) \tag{8}$$

Here, the asserting AS_x creates the ATT_{ASx} using the proof element *PE*, its id AS_x, a timestamp TS_{ASx}, and encrypts the information with AS_x's private key.

Subsequently, the requesting *AS* receives all the *ATT*s from all asserting *AS*s, and forwards the *ATT*s to the *AP*. The *AP* then saves the *ATT*s along with the *TE* for future audit. This completes an asserted accountable proof collection phase.

E. Audit Proof Collection

The auditor communicates with the *AS*s to validate an asserted proof. We call this phase the audit collection protocol, where the auditor communicates with the corresponding *AS*s to collect the timing information. The sequence diagram for the audit collection protocol with a proof configuration of three servers is illustrated in Figure 2.

Initially, the auditor is presented with the *ATT*s and the *TE* of a proof by the *AP*. The auditor then validates the signatures in the *ATT*s for the given *AS*s who have asserted the proof.

The auditor then extracts *PE*s, and *TS*s from the *ATT*s and matches all the *PE*s of them to be same. The extracted *TS*s are later used for validation.

Once successfully verified, the auditor then extracts the *HTE*, and *GUID* from the *PE*. Additionally, the auditor generates a hash HTE_{Aud} of the *TE* provided by the *AP*. The *GUID* and TS_{AP} are also extracted from the given *TE*. The extracted values from the *TE* are then compared to the *HTE* and the *GUID* obtained from the *PE*. If successfully verified, the auditor saves TS_{AP}, which is used later in the validation phase.

The auditor then takes the individual *PE*s (obtained from the *ATT*s) and requests the *AS*s for an assertion on the corresponding *PE*s. Each *AS* matches the received *PE* with the list of previously stored *PE*s, and replies with a new attestation with the same parameters as shown in expression 8.

Finally, the auditor receives the fresh *ATT*s, extracts the time stamps, and validates the proof. The validation scheme is defined next, where the auditor verifies the claim of possession of the data item by the given *AP*.

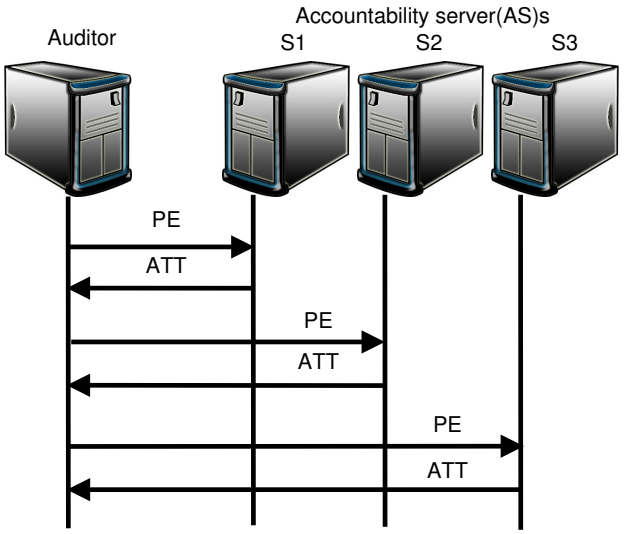

Fig. 2. The sequence diagram presenting communications between an auditor, and accountability servers(AS) of an audit proof collection for a proof configuration of 3 servers.

F. The Accountability Validation Scheme

In this section, we describe the process of validating a claim of ownership of data using the asserted accountable proof. The auditor is required to calculate an aggregated trust score for a given proof. Thus, we define an aggregate trust score (σ) as the combined effective trust score of the *AS*s for a given proof.

To calculate the aggregate trust score, the auditor calculates the trust loss factor (ψ) and the effective trust (Υ) for each *AS*s provided in the given proof.

Initially, we calculate the time differences between the request timestamps and the assertion timestamps, Δ_{AsP} and Δ_{AdP}, for both asserted proofs and audit proofs. Then, we calculate the combined median τ, for (Δ_{AsP}, Δ_{AdP}). The trust loss factor ψ for each *AS*s is then calculated as follows:

$$\psi_i = max\left(0, \frac{max(\Delta_{AsP,i}, \Delta_{AdP,i}) - \tau}{\tau}\right)$$

Next, we calculate effective trust contribution for each server as follows:

$$\Upsilon_i = \delta_i \times min\left(1, f(\psi_i)\right)$$

$$f(\psi_i) = \frac{1 - log_{10}(\psi_i + 0.1)}{1 - log_{10}(0.1)}$$

Here, the function $f(\psi_i)$ can be chosen as required. In our case, we used the above definition for $f(\psi_i)$ because of its sharp fall with a small increase in the value of ψ_i.

In this context, we define trust depreciation as the amount of trust reduced from the initial trust score δ provided within the proof configuration, to the effective trust contribution Υ. From the above formulation, we ensure that trust depreciation

is greater for *AS*s with elevated/higher trust scores, compared to *AS*s with the evenly distributed trust scores.

Subsequently, we calculate the aggregated trust score σ, where $\sigma = \sum_1^n \Upsilon_i$. The proof is thus accepted if the trust threshold is valid, that is $\sigma \leq \xi$, or discarded otherwise.

Finally, the auditor reports a timing accountability for the proof of ownership presented by the *AP*. The auditor presents the validity of the claim within the range ($TS_{AP} \pm \tau$). Hence, the possession of the data by the given *AP* can be assumed to be true within the stipulate range for the time bounded within ($TS_{AP} \pm \tau$).

IV. System Evaluation

A. Threshold Optimization

We allow jitters upto 50% of the τ in either side. Thus the allowable ψ_i is less than 0.25 ($\pm 25\%$). In such case the minimum calculated aggregate trust score σ can be 0.728. Since, this is the worst case scenario, we set the threshold at 0.75. A stringent security configuration will have higher threshold depending on allowance in jitters. The selection of function in calculating trust score is arbitrary. We choose it because of sharp decline for increase in x values. We can choose a different function, and set the threshold accordingly.

Data ownership at a certain time is called an event. If aggregated trust score, σ, is greater than security configuration threshold, ξ, the auditors calculates the event time by adding τ to the user reported time of the event.

B. Detecting Collusion

Let us consider a security configuration of n servers. The application service provider(AP) is colluding with not more than p server to setup a different event timing where $p = \lfloor \frac{n-1}{2} \rfloor$. In such case, there is always at least $n - p$ ($> p$) trustworthy servers, and the timing difference median(τ) will always be contributed from one the trustworthy servers. Since, all trustworthy servers will work honestly, their trust loss factor will be near 0. Hence, they will contribute to the aggregated trust score with minimal trust depreciation. On the contrary, since p colluding servers fails to setup τ in their favor their trust loss factor will be high. Thus effective trust score of colluding servers will be low, as effective trust score calculation reacts sharply towards higher trust loss factor. Now, the validation will either fail due to less than the trust score threshold(ξ) when many servers are colluding, or it will validate ξ with correct event timing. Here colluding servers are effectively filtered as outliers. This makes the system tamper evident against collusion of upto p servers in a security configuration of n servers.

Here, we analyze our system against various type of attacks. We will present how different attacks in threat model are mitigated though our system.

C. Security Analysis

In this section, we specify the different possible attacks on our system, and present the reasoning how our proposed scheme is resilient to the given attacks.

- **False or Cloned Proof**: This attack arises when an application service provider (*AP*) can generate a forged proof of a document. An alternate attack will be utilizing an existing valid proof to represent a different document. We show that in our system these attacks are not possible. Since each proof is associated with a GUID (5) and the proof element (7), and the attestations are signed with private keys (8), they are tamper evident. Also, the timing element which has the original data matched with a GUID is signed with a private key before generating the proof element and attestations. So, the *AP* can not later modify it. Furthermore, the GUID is issued by an accountability server, and the proof element is validated by all the signing accountability servers during the audit. Hence, it is not possible for a client to generate a forged proof or modify an existing valid proof to represent a different document.
- **Time Tampering**: In this attack, the application service provider uses an otherwise valid proof of a document but attempts to tamper the timing value represented by it. As all attestations are signed by servers with respective private keys (8), when the *AP* modifies the time stamp in the timing element (5) the proof element (7) changes in the proof. Proof element in the attestations will not match the tampered proof element. Hence, it is not possible for the *AP* to maintain the attestations with valid signature in such case. An auditor will detect the tampering and discard such proof as invalid. Thus our system prevents such attack.
- **Mutual Collusion**: For mutual collusion attack, the application service provider colludes with other accountability servers to certify a earlier time stamp in the proof compared to the time stamp it actually represents. As we have proved in subsection IV-B, such collusion will either produce valid timing or the collusion will result in invalidating the proof by auditor provided collusion is limited to p servers in a security configuration with n accountability servers. Hence, the auditor will always report either a valid timing or invalidity of the proof.
- **Confidentiality Breach**: For such attack, the attacker tries to gain hold of the application service provider's data sent for proof. We observe in Figure 1 no element derived from the data is transferred before the mutual authentication, and the setup of shared key. Moreover, after these two steps each communication is encrypted using the shared key. Therefore, an external attacker can not learn the content of the data or part of it in any of the steps of our system through intercepting the communications between the entities.
- **Leaking Privacy**: A credible system must not be susceptible to leaking information. Even if the attacker can not get hold of the direct information such as the data, they may target to acquire additional knowledge of valuable information, such as usage pattern. In case of both attestation proof generation or audit, the original

data never leaves the requesting accountability server. All other communications are made using *PE* which only reveals the *GUID*, and the AS_{id}. This *GUID* uniquely identifies an attestation proof and its corresponding audit proof for a particular accountability server. But since it is *AP* invariant there is no direct link to the *AP*. Hence, compromising any attestation proof generating accountability server will not reveal information about the requesting *AP*.

- **Auditor Falsification**: The auditor can also attack the system. He can be compromised and may try to produce invalid proof to frame the *AP* with wrong time stamp. If the auditor colludes with not more than p servers in a security configuration with n accountability server, as per subsection IV-B, it is not possible to produce a valid proof with incorrect timing. An auditor can also try to give invalid final judgment to an otherwise valid attested proof. This is a denial of service attack (DoS) and the user can request for a different auditor.

V. Evaluation of Results

For evaluating our design and to demonstrate the proof of concept, we built a prototype with Java Programming Language. We chose Java Programming Language for the convenience of network programming, and out-of-the-box support of public-private key encryption. We utilized JSON [11] as the encoding for data transfer. JSON is a lightweight, easy to use, platform independent encoding and available in most programming languages. We simulated different error, and attack scenarios and our prototype caught those successfully. We also conducted some experiments to validate performance of our system. Our design focus of the system was scalability, and efficiency. We show that our system is capable of in both aspects. We ran our system in small instances of Amazon EC2[1].

A. Data Overhead Independence

For a server-client architecture in a cloud systems there will be many proof attestations from each clients. Hence, minimal storage requirement is an important issue. In this experiment, we show that additional data required for preserving timing element proofs is independent from the original data size, and they scale over number of requesting servers linearly. This is of significant importance as in such system increase in clients will not affect data overhead requirement for other clients. Also original data is not duplicated over the proof elements. This facilitates keeping the overall storage required minimum for each proof and the overhead is limited to number of servers in a proof configuration only. This result is shown in Figure 3. The graph presents three series of plots for 512, 1024, and 2048 bytes of original data. The linear relation between total proof size vs number of attesting servers indicates storage scalability in terms of attesting servers. Also, parallel difference between series implies independence of overhead with original data.

[1]Amazon EC2 Instance Types, http://aws.amazon.com/ec2/instance-types

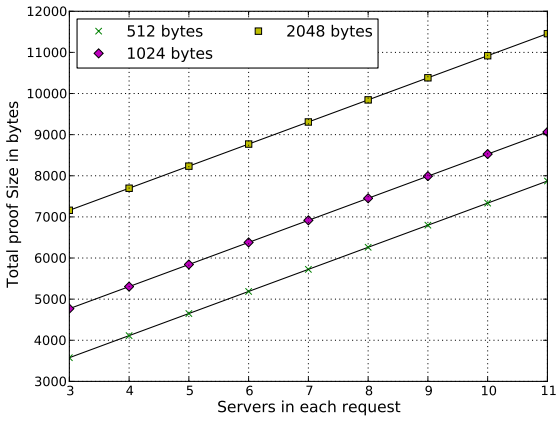

Fig. 3. A plot of number of attesting servers vs total proof size required for each request. The proof size is shown in three series of plots for 512, 1024, and 2048 bytes of original data. This graph represent the linear relation between total proof size vs number of attesting servers. Also, parallel difference between series implies independence of overhead with original data.

B. Scalability

Scalability is critical for any system to be applicable in clouds. In Figure 4, we illustrate the relation between data size and throughput for varying number of servers. It can be seen that increasing the total number of servers increases the throughput consistently. For change in data size, throughput of the system is affected little except when data sizes are very small and total servers in the system is very small. From this, we argue that our system is scalable for large number of servers to serve arbitrarily large number of requests. Alternatively, we can handle increasing number of clients by increasing the number of servers.

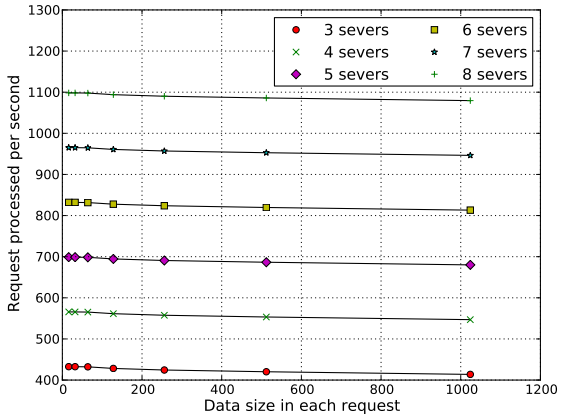

Fig. 4. A plot of data size vs throughput for different number of servers. The consistent increase in throughput indicates the scalability of the system. Also, series plots are near parallel to x-axis denotes little effect on the throughput due to increase in data size.

VI. Related Works

Though history preservation through accountable timing and auditing are very useful and widely used in distributed system, there is a very limited amount of related works in this topic in cloud computing. The system presented by Maniatis et al. [14] comes close to our work. They proposed a mechanism to create a tamper-evident historic record or secure timeline of different persistent authenticated dictionaries. The records are maintained by mutually distrusting servers to protect the historic integrity in a distributed system. In contrast, our proposed scheme preforms the task of preserving accountability using a timing element for data in the cloud. The same authors presented a system in FAST 2002 [13] using digital signatures for storage of files over long periods of time. However, it does not incorporate timestamping and re-timestamping of the archive.

Other recent research in this area is concerned with provable secure timestamping services [3, 5]. Buldas et al. [4] presented a timestamping scheme with an additional audit functionality using universally one-way hash functions, which is a weaker property than collision resistance. Hasan et al. [10] proposed a very relevant work on preventing secure history forgery. However, their scheme contrasts with our approach as they focus on the provenance of data rather than a proof of ownership. Haber et al. [8] used timestamps of digital data to maintain the integrity and authenticity. Bleikertz et al. [2] proposed a different technique to audit the multi-tier architecture in the cloud using reachability and attack graph. Additionally, Haeberlen [9] proposed a theory-based scheme to verify the accountability with respect to particular agreements both in cloud provider and customers. Our system differs from these works not only for implementation on a real cloud system, but also we present a time range for the validity of a claimed ownership, which is not present in these approaches.

Public auditing for cloud data with data privacy was proposed by Wang et al. [18]. The proposed scheme allows a third party auditor to verify the integrity of publicly available data in a cloud. Additionally, to allow the auditing tasks for multiple users at the same instant, they extend their mechanism to empower batch auditing by leveraging aggregate signatures. However, the scheme innately assumes the provider not to be malicious, does not provide any options for private data, and is not fairly scalable to be applicable in cloud architectures. Sundareswaran et al. [17] proposed a decentralized information accountability scheme to keep track of the actual data usage of JAR files for users in the cloud. Ryan et al. [12] discussed a five layered framework named TrustCloud for cloud accountability. It attempts to increase trust in cloud service providers through accountability and heightened data transparency. However, we utilized an aggregated trust score which is based on an effective trust contribution by each server involved in validating the accountable proof. Furthermore, we claim that our approach can be relaxed with realistic restrictions.

VII. Discussion

In this paper, we designed a secure system for accountable proof of ownership through timing element. We focused our design for the proof generation, and auditing the proof to be fast, and isolated so that they can be applied to the scale of cloud. In return, we relaxed the required precision of time, which in effect relaxed the chaining of events to the extent of 2τ timing window. This gave us a coarse resolution of event timing and an implicit loose chaining of data ownership or events.

Keeping accountable history has historically been an important issue for mankind. In this paper, we mentioned an infamous controversy regarding the invention of calculus and also referred to one hypothetical current issue. From Figure 4, we argue in favor of our designed system to be scalable. Our experimental result in Figure 3 shows that history data size increases almost linearly with increase in number of servers in a request. In our proposed system, clients and servers both keep their own local history for future use which helps keeping load off the servers at the same time facilitate audit when client looses part of the proof. Also, we showed in Figure 3 that, for our proposed system, data overhead is not dependent for original data size and only depends on number of servers participating in the proof. Additionally, proof storage overhead is independent of number of application service providers in the system as each proof is independent of other application service providers' proofs.

There are some notable limitations of our system. The timing of ownership generated from an acceptable proof has an window of $\pm\tau$. Hence, every ownership timing has an window of 2τ which may overlap with others. We used GUID in our system but we did not discuss about the distribution control of GUID. In our system, Denial of Service (DoS) attack from the client is not discussed. We also did not implement long term secure storage for the signed proofs.

VIII. Conclusion and Future Work

We presented a novel system of secure timing element based attestation for accountability. Our system design focused on the scalability of cloud services. As from our evaluations of the system in section V, we argued that our system can be successfully implemented for large scale cloud based services.

In this system, we use original data, or data hash as our input. We plan to extend our works using Merkle Hash Tree (MHT) [15], Distributed Hash Table (DHT) [1] or Authenticated Distributed Hash Table (ADHT) [16] and check the efficiency of new system and provide the comparison between them. We use timing entanglement in our system. We want to extend our system using timeline entanglement and compare the performance with this system. We leave as open problems the design of authenticated distributed data structures for more general queries and additional security issues in the model, such as denial of service (DoS) attacks and other Byzantine behaviors. Furthermore, the design and implementation of the proof request is atomic. This can be extended to batch mode to

support a group of request at the same time. This may reduce total packet transfer in the system greatly.

ACKNOWLEDGEMENTS

This research was supported by a Google Faculty Research Award, the Office of Naval Research Grant #N000141210217, and the Department of Homeland Security Grant #FA8750-12-2-0254.

REFERENCES

[1] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *Communications of the ACM*, 46(2):43–48, 2003.

[2] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *Proceedings of the ACM Cloud Computing Security Workshop '10*, pages 93–102, Chicago, Illinois, USA, October 2010.

[3] A. Bonnecaze, P. Liardet, A. Gabillon, and K. Blibech. Secure time-stamping schemes: A distributed point of view. *Annals of Telecommunications*, 61(5):662–681, 2006.

[4] A. Buldas, P. Laud, M. Saarepera, and J. Willemson. Universally composable time-stamping schemes with audit. In *Information Security*, volume 3650 of *Lecture Notes in Computer Science*, pages 359–373. Springer Berlin Heidelberg, 2005.

[5] A. Buldas and M. Saarepera. On provably secure time-stamping schemes. In *Advances in Cryptology - ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 500–514. Springer Berlin Heidelberg, 2004.

[6] BusinessWare. Gartner reveals top predictions for it organizations and users for 2012 and beyond. Online at http://bit.ly/11qxVVV [Accessed on March 22, 2013].

[7] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[8] S. Haber and W. Stornetta. How to time-stamp a digital document. In *Advances in Cryptology - CRYPT0 '90*, volume 537 of *Lecture Notes in Computer Science*, pages 437–455. Springer Berlin Heidelberg, 1991.

[9] A. Haeberlen. A case for the accountable cloud. *ACM SIGOPS Operating Systems Review*, 44(2):52–57, 2010.

[10] R. Hasan, R. Sion, and M. Winslett. The case of the fake picasso: preventing history forgery with secure provenance. In *Proceedings of the USENIX Conference on File and Storage Technologies '09*, pages 1–14, San Francisco, CA, USA, February 2009.

[11] JSON. Introducing json. Online at http://json.org [Accessed on March 22, 2013].

[12] R. K. L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee. Trustcloud: A framework for accountability and trust in cloud computing. In *Proceedings of the 2011 IEEE World Congress on Services*, pages 584–588, Washington, DC, USA, July 2011.

[13] P. Maniatis and M. Baker. Enabling the archival storage of signed documents. In *Proceedings of the USENIX Conference on File and Storage Technologies '02*, Monterey, CA, USA, January 2002.

[14] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium '02*, pages 297–312, San Francisco, CA, USA, August 2002.

[15] R. C. Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO '89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer New York, 1990.

[16] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. *Peer-to-Peer Systems*, pages 261–269, 2002.

[17] S. Sundareswaran, A. Squicciarini, D. Lin, and S. Huang. Promoting distributed accountability in the cloud. In *Proceedings of the IEEE 4th International Conference on Cloud Computing '11*, pages 113–120, Washington DC, USA, July 2011.

[18] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *Proceedings of the 29th Conference on Information Communications '10*, pages 525–533, San Diego, California, USA, March 2010.

[19] A. R. Yumerefendi and J. S. Chase. Trust but verify: accountability for network services. In *Proceedings of the 11th workshop on ACM Special Interest Group on Operating Systems European Workshop '04*, Leuven, Belgium, September 2004.