# Building Trust in Storage Outsourcing: Secure Accounting of Utility Storage

Vishal Kher and Yongdae Kim
Computer Science & Engineering, University of Minnesota
{vkher,kyd}@cs.umn.edu

## Abstract

*We are witnessing a revival of Storage Service Providers in the form of new vendors as well as traditional players. While storage outsourcing is cost-effective, many companies are hesitating to outsource their storage due to security concerns. The success of storage outsourcing is highly dependent on how well the providers can establish trust with their consumers. While significant work has been done to ensure confidentiality, integrity, and availability of data, a practical solution for accounting of outsourced storage is still at large missing. This paper presents Saksha, a secure accounting system that enables automated and verifiable metering of the resources utilized by the consumers. A provider that includes Saksha as a part of its storage service can prove to its customers the amount of resources utilized by them. As a result, Saksha will help to enhance trust by preventing any inflation or deflation of the service usage. Saksha is not restricted to any particular pricing model; it can be applied to the popular pay-per-use pricing model for utility storage as well as many of its variants. In addition, it can be used by the consumers to periodically evaluate their usage and reassess their outsourcing requirements. Saksha is developed such that it can be layered on the top of networked file systems. Our performance results demonstrate that Saksha is efficient and can be used in practice.*

**Keywords:** Trust, Verifiable accounting, Metering, Storage outsourcing, Storage security

## 1. Introduction

Storage Service Providers (SSP) sell storage, bandwidth, and management services to companies that do not have enough money, space, or technical expertise to build and manage their own data centers. We have witnessed a revival of SSPs in the form of new vendors as well as traditional players [11, 8]. A Storage Service Consumer (SSC) subscribes to a SSP and users belonging to the SSC (SSC employees or customers in case the SSC is itself a provider) use the file systems hosted by the SSP to store and retrieve the data. The storage outsourcing pricing models are still nascent. One of the models that is gaining popularity is pay-per-use pricing where consumers are charged based on their usage and disk consumption [3, 7, 1]. For example, Amazon and HP charge their customers for every gigabyte transferred and stored [1, 3] on a monthly basis. Alternatively, in the future, one can use variations of pay-per-use pricing models, such as history-based pricing where a SSP can charge a SSC based on the previous month's usage, or charge more for bandwidth than for storage [22].

While outsourcing storage is cost-effective, many companies are hesitating to outsource their storage mainly due to trust and security concerns. Eliminating these concerns and establishing trust with the Storage Service Consumer (SSC) is pivotal for the success of storage outsourcing. An important question to be addressed in this setting is - how can consumers trust providers? More precisely, in the context of this paper, how can a SSC trust a SSP to report the correct usage values and charge it the right amount? A greedy SSP can attempt to cheat the SSC by inflating the usage values; whereas on the other hand, a greedy SSC can try to cheat by deflating or denying its usage. Such incidents have been evident in the past [4, 29], which not only went undetected for several years, but also allegedly resulted in a loss of hundreds of millions of dollars and tedious litigations.

A SSP will be able to gain its consumers trust by including automated and secure accounting as a part of its service. *Automated accounting* will enable consumers (or providers) to monitor, view, and predict their usage (or contribution) in a timely manner, whereas *secure accounting* - a mechanism by which a SSP can prove the amount of utilized service to the SSC - can prevent deception and resolve disputes. In current practice, the accounting seems to be primarily based on some estimates such as using logs (which can be manipulated) and trust rather than provable evidences. In addition to secure accounting, the SSP should provide services, such as confidentiality, integrity, and availability. While most of the recent research related to secure outsourcing is centered around ensuring confidentiality [15, 16, 21], integrity, and availability [26] of outsourced data, a practical solution for secure and automated accounting is still at large missing.

In this paper, we present *Saksha* - a system that enables automated and secure accounting of the services utilized by the consumers. By including Saksha as a part of the storage service, the SSP will be able to measure and give a non-repudiable, publicly verifiable proof of the amount of storage and bandwidth utilized by its consumers. The SSP can include these proofs as a part of its periodic billing cycle. The SSC can verify these proofs and rest assured that it is indeed paying for the services utilized by them. Saksha ensures that a consumer cannot deny its usage and a provider cannot inflate its consumer's usage. Each party can detect any misuse and resolve disputes in a timely manner without requiring any third party intervention, which is often time consuming and expensive. To the best of our knowledge, Saksha is the first secure accounting system for utility storage that can be layered on top of networked file systems. In this paper, we also present a generic protocol for secure accounting, an instance of which is included in Saksha. Our experimental evaluation demonstrates that Saksha is efficient and incurs a small overhead during data path.

## 2. Design Criteria

### 2.1 System Model

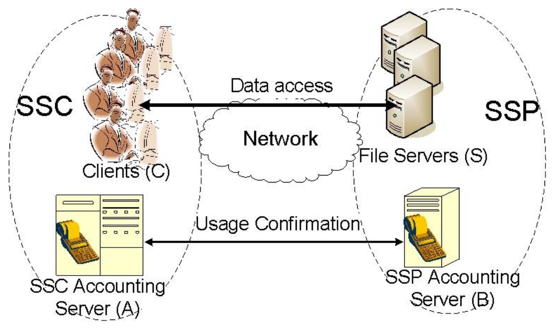


**Figure 1. System model overview.**

Figure 1 gives a high-level overview of our system. It consists of four entities namely, the SSC client machines, the SSC accounting server $A$, SSP file servers, and the SSP accounting server $B$. The SSC client and the SSP file server is denoted as $C$ and $S$, respectively. The SSC clients mount remote file systems to store and retrieve data from $S$.

A client $C$ attests the fact that it has consumed some resource from $S$ by giving a verifiable *receipt* to $S$. $S$ verifies this receipt and deposits it at $B$. Receipts from all the file servers are collected by $B$ and sent periodically to $A$, who then verifies these receipts and uses them to calculate the amount of resources consumed by $C$ (and all other clients). These receipts stand as proof-of-work and $B$ can use them to prove to $A$ or any third party (in case of disputes) the amount of resources consumed by its clients. In case $C$ fails to give receipts to $S$ (i.e., $C$ attempts to cheat), $S$ can stop providing any service to $C$ until $C$ gives the receipts. We would like to stress that in figure 1, the SSC accounting server $A$ is shown as a separate server machine for simplicity. In reality, it is just a simple, light-weight process that can be run on an average machine and does not incur any significant maintenance overhead.

One way of generating receipts is to use RSA digital signatures. For example, every time $C$ consumes $y$ bytes of storage space from $S$, $C$ can give a signature on tuple $\langle C, S, y \rangle$ to $S$. $S$ can verify the signature and deposit all such receipts at $B$, who in turn will forward the receipts to $A$ for accounting. The first problem of this approach is that it will impose significant computation overhead on $A$, since $A$ has to verify all the receipts from all of its clients. In addition, $B$ will also have to verify these signatures to resolve any disputes. This verification cost can be overwhelming, especially on $B$, since a SSP can have associations with many consumers. The second problem of this approach is that it requires public key operations (on $C$ and $S$) during file access, which can introduce additional latencies in the data path. Therefore, to avoid these problems, Saksha must use receipts that will satisfy the following requirements: 1) generation and verification of receipts must be efficient, 2) they must be publicly verifiable and non-repudiable, 3) each receipt must have a unique sender and receiver identifier, otherwise two different recipients (e.g., servers) will be able to claim credits for the same receipt, and 4) no one should be able to replay the receipts.

### 2.2 System Requirements

**Resistance to abuse**: The SSP must be able to provide a proof of its clients consumption. The SSP should not be able to inflate its customer's usage; neither should the customer be able to deny its consumption.

**Timely detection of misuse**: Saksha must be able to detect any misuse of the system (inflation/deflation) as soon as possible. Timely detection of greedy behavior from either of the party will enable the other party to take appropriate actions and minimize resulting losses.

**No third party intervention**: Since involving a third party is often too expensive and increases the system complexity, the accounting system must not require any intermediary.

**Generic mechanism**: The accounting mechanisms must be independent of the underlying file systems so that they can be easily integrated with different file systems.

**Minimized computation and communication overhead**: The computation, communication, and storage overhead on all of the entities should be as small as possible. Adding computation and communication overhead during data path will increase the latency of reads and writes and reduce the performance of the system. As a result, it will hamper the deployment of the accounting mechanism.

### 2.3 Assumptions

For the purpose of entity authentication, we assume presence of a certificate authority (CA) and that every client machine ($C$) and file server ($S$) has a public-key certificate.

We assume that the SSCs do not trust usage values reported by the SSPs. In other words, the SSPs can attempt to inflate the usage values. For instance, by simply reporting wrong usage values at the end of the billing cycle, or by self-creating random files and storing them in the consumers storage space. On the other hand, SSCs can attempt to deny their usage. It is assumed that the SSC trusts all its employees and there exists a mutual trust between SSC clients and the accounting server $A$. Similarly, there exists mutual trust between $B$ and $S$. Also, we assume that SSC employees do not collaborate with the SSP. Such collaboration attacks can never be prevented, since a SSC employee can simply keep consuming the resources.

We assume that the underlying file system will provide the necessary security features, such as authorization, confidentiality, etc. For simplicity, we assume that the consumers use mechanisms to ensure integrity and freshness of their data. Without such mechanisms in place a server will be able to inflate the usage values by manipulating the contents of the files. For example, by ignoring end of file (EOF) during read operations and instead providing some garbage data, or a server can perform rollback attacks [19] by replacing the current version of file by an older version of smaller size so that the server will get more receipts during the following writes. In another attack, the server simply drops the client's file; thus, claiming credits for storage without actually contributing for it. Such data corruption attacks, can be prevented using any of the existing techniques, such as [19, 30, 26, 21]. In this paper, we focus only on verifiable accounting and assume that the clients are using appropriate data integrity mechanisms.

Finally, we assume that the client gives receipts to the server for the utilized service and vice versa. This is a reasonable assumption since both the SSC and the SSP are interested in a long-term relationship. Failure to give receipts is a clear indication of an attempt to cheat.

## 3. Design

In a secure accounting system, we need to 1) define the type of receipts, 2) measure the amount of resources consumed by the clients, 3) define the unit of accounting, and 4) design a secure accounting protocol.

### 3.1. Verifiable Receipts

Saksha is primarily focused on measuring two types of resources - bandwidth and storage space. Unlike bandwidth which is a one-time resource, storage space can be reduced due to deletes. Therefore, accounting of storage service includes keeping track of three activities: bandwidth consumption, increase in storage space, and decrease in storage space. In order to support verifiability of these three operations Saksha defines three types of receipts (one for each): `bandwidth_receipt`, `storage_receipt`, and `delete_storage_receipt`. The former two are given by $C$ to $S$ whenever $C$ consumes more storage space or transfers data from $S$. Since these receipts are non-repudiable, $S$ uses these receipts to claim credits. To avoid the situation where $S$ attempts to inflate storage consumption, a reverse operation must happen for delete - $S$ should give a `delete_storage_receipt` to $C$.

### 3.2. Measuring Resource Consumption

**Bandwidth measurement** In the context of this paper, bandwidth is defined as the amount of data transferred between a client $C$ and a server $S$ within the accounting interval. For example, the bandwidth consumed due to a read request is the size of read request plus the size of server response. The accounting interval is expected to be at least an hour and in most of the cases a day, since SSP charge their customers based on consumptions per day or per month [3, 7, 1]. In Saksha both $C$ and $S$ keep track of the bandwidth used by $C$ by measuring the amount of bytes transferred between $C$ and $S$ (due to operations such as read, readdir, write, etc.).

**Storage Measurement** In order to give a `storage_receipt` to the server, the client has to predict how much new disk space will be consumed as a result of the current `write` operation. In reality, only the file server can accurately measure the disk usage. A write by a client may not always result in increase in the disk space. For example, a client overwrites some of its old data, which results in data transfer but does not result in increase in disk space. On the other hand a write may result in fragmentation at the server file system; thus consuming more space. Further, the server file system may accumulate the tails (last blocks) of each file and store them collectively in special purpose blocks. In this case, a write may not consume more disk space since the data might get squeezed into some previously allocated blocks. Thus, a client sitting outside the file server cannot predict accurately the result of its write operation. Saksha resolves this problem by measuring the amount of "additional data" uploaded/added by the client during the write operation instead of measuring the actual disk usage. For example, suppose the current file size is 6KB and the client writes additional 3KB starting from offset 4096, then the client consumes 1KB of additional storage space. Based on the current file size, the offset, and the size of current write, the client can calculate the amount of additional upload ($y$) by using the following simple equation.

$$y = (offset + write\_size) - current\_file\_size$$

From the client's perspective it is uploading $y$ new bytes (if $y$ is positive; if $y$ is negative it means that the client has overwritten a portion of the file) to the file server, and that is the amount of additional storage consumption. How the file server stores these $y$ bytes is extremely difficult to predict and can change if the server deploys a different file system. Besides, from the SSP's perspective, accurately measuring storage space consumption at the granularity of disk block may not be necessary as long as it can accurately measure the amount of additional content uploaded by the client. The difference between the size of the uploaded content and the actual disk usage can be considered as insignificant overhead. Thus, for the remainder of this paper, when we say client consumes $y$ amount of additional/new storage space, it means that the client has uploaded $y$ bytes of additional content.

Saksha measures the amount of content deleted by the client by monitoring file system operations that eventually result in reduction in storage space, such as `unlink`, `rmdir`, or `truncate`. To account for the delete space, `delete_storage_receipts` are given by $S$ to $C$. In a file system, in addition to write and delete operations, space is also consumed due to metadata operations by creating a file (e.g., `mknod`) or by setting attributes (e.g., `setxattr`). The amount of storage space consumed by the `mknod` operations is difficult to predict, since it also depends on the actual file system block size. Saksha approximates this increment by assigning a flat cost for the file creation operations. Storage space consumed by attributes is handled similar to the write operation discussed above.

## 3.3. Unit of Accounting

During accounting for storage and bandwidth, presenting verifiable accounting information for every byte increase in storage space or every byte transferred will be expensive and is unwarranted. In Saksha, accounting is performed for a bigger collection of data called a *chunk*. For example, in the case of bandwidth, we measure the number of chunks transferred between $C$ and $S$ during the accounting interval. Similarly, in the case of storage, we measure the number of additional chunks uploaded by the client during the accounting interval. The size of a chunk is part of a policy decision and should be agreed upon by both the parties (it can simply be a part of the service level agreement). Thus, in Saksha, a receipt is exchanged only when the client consumes a chunk of storage/bandwidth or deletes a chunk.

## 3.4. Accounting Protocol

In this section, we begin by presenting a generic framework for storage utility accounting. This framework can be instantiated into different protocols that primarily vary in the ways of generating, exchanging, and verifying receipts. We then present the details of the Saksha accounting protocol, which is an instantiation of this generic structure.

**3.4.1. A generic protocol structure.** The storage service accounting schemes follow the structure shown below:

**Initialization** The SSC accounting server $A$ and the SSP accounting server $B$ generate their own public-private key pairs denoted by $(Pk_A, Sk_A)$ and $(Pk_B, Sk_B)$, respectively, and get their public-key certificates from a trusted CA. Let $Cert_A$ and $Cert_B$ denote the certificates for $A$ and $B$ respectively. Each client[1] $C$ generates its own public-private key pair and acquires a public-key certificate $Cert_C$ from $A$, which is signed using $Sk_A$. $Cert_C$ binds $Pk_C$ to $C$ and symbolizes the fact that according to $A$, $C$ belongs to $A$'s organization. In addition, $A$ can optionally include any policy information in $Cert_C$. Similarly, each server $S$ acquires a public-key certificate $Cert_S$ from $B$.

**Beginning of the accounting period** During this phase the clients and servers can pre-generate receipts based on the expected daily usage. While this pre-generation of receipts is optional, it will greatly reduce the computation effort during data access. The accounting period $t$ is expected to be at least an hour and in most cases a day, since SSPs charge their customers based on consumptions per day or per month [3, 7, 1].

**Increase storage request** This step is performed when $C$ consumes new storage space from server $S$ due to successful file system operations, such as `write,mkdir`, `setxattr`, etc. In this step, $C$ sends $k$ `storage_receipts` to $S$ if it consumes $k$ chunks of storage space from $S$. $C$ maintains a storage usage counter $X$ that counts the amount of storage space consumed by $C$ since the last transfer of a `storage_receipt` to $S$. $C$ sends $k$ receipts (or one receipt of value $k$) to $S$, where $k = \lfloor X/b \rfloor$ and $b$ indicates chunk size. After transferring receipts, $C$ then decrements $X$ by $k * b$. After receiving the receipts, $S$ verifies them and stores them locally (or sends them to $B$ for accounting). Note that since a write operation also consumes bandwidth, this step will also invoke the bandwidth request phase. Exceptions to this are file system operations that result in increase in storage space, but do not necessarily transfer data (pre-allocation operations such as `truncate`).

**Bandwidth request** This step is performed every-time $C$ transfers some information or receives some information

[1]Client here indicates client machine.

from the server. $C$ maintains a bandwidth consumption counter $Y$ that counts the number of bytes transferred between $C$ and $S$ (due to any file system operation). Similar to the above step, $C$ sends $k$ receipts (where $k = \lfloor Y/b \rfloor$) to $S$ and decrements $Y$ by $k*b$. $S$ verifies the received receipts and stores them locally or sends them to $B$ for accounting.

**Decrease storage request** This step is performed when $C$ deletes some storage space from $S$ (e.g., due to `truncate`, `unlink`). $S$ maintains a delete usage counter $Z$ that counts the amount of storage deleted by $C$. If $k = \lfloor Z/b \rfloor > 0$, $S$ gives $k$ `delete_storage_receipts` to $C$ and decrements $Z$ by $b * k$. $C$ verifies these receipts and store them locally.

**Usage measurement** At the end of the accounting period, servers send their receipts to $B$. $B$ counts the usage, keeps a record of accounting interval, and stores all of the receipts as evidence. The receipts should be kept at least until $A$ and $B$ agree with the usage values. Note that $B$ does not have to verify the receipts until a dispute since $S$ has already verified them. $B$ then forwards these receipts to $A$. Since $B$ has no incentive to forward the delete receipts to $A$, $C$ must forward the delete receipts received from $S$ to $A$. Finally, $A$ verifies these receipts (if necessary) and calculates the storage ($U_s$) and bandwidth ($U_b$) usage in terms of chunks during interval $t$ as follows:

$$U_s = \#storage\_receipts - \#delete\_storage\_receipts \quad (1)$$
$$U_b = \#bandwidth\_receipts \quad (2)$$

**3.4.2 Preliminaries.** Before we explain our protocol in detail, we give an introduction to hash chains, which is a primitive of our protocol. Hash chain is a popular cryptographic technique that has been used before for one-time passwords [18], one-time signatures [17], micropayments [5, 25, 24], or to measure web clicks [6]. Let $h$ denote a cryptographically strong hash function such as SHA1. A hash chain is generated by recursive application of $h$, where the input to the hash function is the output of the previous iteration. A hash chain is created in reverse order as follows:

$$r_m = x,\ r_j = h(r_{j+1}),\ h^m(r_m) = r_0,$$

where $x$ is a random seed, $j = m - 1, \ldots, 1, 0$, and $r_0$ is called the root of the hash chain.

A hash chain can be used for accounting as described below. Suppose $C$ wants to read $k$ chunks from server $S$. $C$ will generate a hash chain described above and create a *commitment certificate* $Cert_C^S = \{C, S, r_0\}_{Sk_c}$, where $\{M\}_{Sk_c}$ denotes a message $M$ signed using $Sk_c$. $Cert_C^S$ is $C$'s commitment to root $r_0$. For $k$ chunks, along with $Cert_C^S$, $C$ gives $k$th value of the chain as a *receipt* to $S$. $S$ verifies $Cert_C^S$ and applies $h$ to $r_k$ recursively $k$ times to get $r_0' = h^k(r_k)$. $S$ then verifies if $r_0'$ is equal to $r_0$ present in $Cert_C^S$. If yes, $S$ has received $k$ receipts for which it can claim credits by presenting $Cert_C^S$ and $\langle k, r_k \rangle$ to verifier.

**3.4.3 Saksha protocol details.** We now describe our protocol in detail.

**Initialization** As described in section 3.4.1 , all entities acquire their respective public-key certificates.

**Beginning of the accounting period** A background process running on $C$ and $S$ called `rcpt_srv` keeps generating hash chains and makes them available for accounting. Since hash chain generation is inexpensive, this process over-provisions the chains to ensure that hash chains are available beforehand. Note, in case a hash chain is not available, it can be easily generated on the fly, which will incur a very small one-time delay.

When $C$ connects to $S$, $C$ acquires two hash chains from the `rcpt_srv`, one of which will be used to give `bandwidth_receipts` and the other to give `storage_receipts`. Let the hash chain used for `bandwidth_receipts` be denoted as $b_k^i, b_{k-1}^i, \ldots, b_0^i$ and the chain used for `storage_receipts` be denoted as $s_l^i, s_{l-1}^i, .., s_0^i$ where $i$ is server identifier and $k$ and $l$ denotes the length of the respective chain. Each hash value in the chain (except root) will be used as a receipt. For each chain, $C$ creates a commitment certificate as shown below:

$$BW_C^S = \{C, S, b_0^S, ''bandwidth'', start, end\}_{Sk_c}$$
$$ST_C^S = \{C, S, s_0^S, ''storage'', start, end\}_{Sk_c}$$

Where, $C$ and $S$ are unique identities of the client and the server respectively, root indicates the root value of the hash chain, and the type of receipt chain (i.e., hash chain) is indicated as: "storage" for `storage_receipts` and "bandwidth" for `bandwidth_receipts`. The period of validity of the chain is indicated by the *start* and *end* fields. Similarly, $S$ acquires one hash chain from its `rcpt_srv` which will be used to give `delete_storage_receipts` to $C$. Let us denote this chain as $d_q^j, d_{q-1}^j, .., d_0^j$, where $j$ is the client identifier and $q$ denotes the length of the chain. The server then generates a commitment certificate for $C$: $D_S^C = \{S, C, d_0^C, ''delete'', start, end\}_{Sk_s}$.

$C$ and $S$ exchange the commitment certificates. Each party verifies the certificates received from the other party and cache the root of each received chain. More precisely, $S$ will cache $\langle C, 0, s_0^S, expiry \rangle$ and $\langle C, 0, b_0^S, expiry \rangle$ whereas, $C$ will cache $\langle S, 0, d_0^C, expiry \rangle$. The commitment certificates are stored in secondary storage.

**Increase storage request** If $C$ has consumed $k$ additional chunks at $S$, then $C$ will send $\langle k, s_k^S \rangle$ to $S$. After receiving this tuple, $S$ will first check the cached expiry and drop this receipt if it has expired. If this check is successful, $S$ will hash $s_k^S$ $k$ times to get $z = h^k(s_k^S)$, and verify that $z$ is equal to $s_0^S$. If yes, $S$ caches $\langle C, k, s_k^S, expiry \rangle$. After, every successful verification of the receipts, $S$ caches the receipt with highest index. Next time, when $C$ consumes $i$ more chunks, $C$ will send $\langle i, s_i^S \rangle$ to $S$ ($i > k$). $S$ will hash $s_i^S$ $i-k$ times and verify if the resulting hash value matches with

$s_k^S$. Thus, $S$ has to perform one hash operation per chunk created by $C$ whereas computation on $C$ is zero (assuming that the receipts are pre-computed).

**Bandwidth request** This part is similar to the "increase storage request" part, except that in this case $C$ will give `bandwidth_receipts` ($b_i^S$) instead of `storage_receipts`.

**Decrease storage request** This part is also similar to the "increase storage request" part, except that in this case, $S$ will give `delete_storage_receipts` to $C$.

**Usage Measurement** In our protocol, this step is performed at the end of each day, to avoid large computations at the end of the accounting period (if the metering period is large e.g., one month). At the end of the day all servers send the last receipts received from the clients as well as the last delete receipt to $B$. Thus, for each client $C$ a server $S$ will send the following information to $B$:

$$\langle Cert_C, ST_C^S, p, s_p^S \rangle \quad (3)$$

$$\langle Cert_C, BW_C^S, q, b_q^S \rangle \quad (4)$$

$$\langle Cert_S, D_S^C, r, d_r^C \rangle \quad (5)$$

Here, $p$ and $q$ indicate the index of the last `storage_receipt` and `bandwidth_receipts` received from $C$. The index of the last `delete_storage_receipt` received by $C$ is denoted by $r$. $B$ stores all of the receipts and estimates the storage increase ($U_s$) and bandwidth usage in terms of chunks for interval $t$ using equations (1) and (2) as follows:

$$U_s = p - r \quad (6)$$

$$U_b = q \quad (7)$$

Similarly, every client sends the following information to $A$: $\langle Cert_S, D_S^C, r, d_r^C \rangle$. $B$ forwards to $A$ information received in (3) and (4). For each client-server pair, $A$ verifies $ST_C^S$ and $BW_C^S$, recursively hashes $s_p^S$ $p$ times and $b_q^S$ $q$ times, and verifies if the corresponding final hash values are the same as $s_0^S$ and $b_0^S$. Similarly, $A$ verifies each of $D_S^C$ and verifies if $d_r^C$ is correct. Finally, $A$ computes the usage using equations (6) and (7).

We stress that *$A$ does not have to verify the hash chains every time* if each $C$ can send to $A$ the index and the value of the last receipt given to each $S$. $A$ can simply compare the values received from $C$ and $B$. If they match, both $A$ and $B$ can exchange signatures agreeing on the usage. However, in case the values don't match, $A$ will have to verify the hash chains that resulted in the dispute. Thus, $A$ has to verify all the hash chains only in the *worst case*. In cases that do not result in a dispute, $A$ will have to perform only two signature generation and verification to agree on the usage metrics with $B$.

# 4. Protocol Discussion and Analysis

## 4.1. Discussion

For simplicity, we have assumed that the chunk size is the same for `storage_receipts`, `bandwidth_receipts`, and `delete_storage_receipts`. In reality, the chunk sizes can be different for each. Since typically bandwidth is consumed significantly more than the storage space, the chunk size for `bandwidth_receipts` can be bigger than the `storage_receipts`. Similarly, since a delete operation removes a bigger collection of data than write operations, the chunk size of `delete_storage_receipts` can also be bigger than that of the `storage_receipts`. The chunk size for `storage_receipts` can be chosen based on the type of data accessed, for example, bigger for multimedia. In sections 4.3 and 6 we show that even with small chunk size of 4 KB or 16KB for each type of receipt, the overhead of Saksha is small.

The computation overhead on $A$ also depends on the accounting interval $t$. In general, for large intervals more time will be spent to perform the hash verifications. If the usage during this accounting interval is also high, the computation overhead on $A$ can be significant. In which case, the usage measurement phase of the accounting protocol can be performed more frequently, for example after every few days. Since $B$ has to send only the last hash of all the chains to $A$, the communication overhead due to this periodic communication between $A$ and $B$ is very low (see section 4.3).

In Saksha, in case $C$ does not give receipts, $S$ will allow $C$ to access files until $C$ utilizes the resources above some threshold without giving any receipts. After the threshold is reached, all requests from $C$ are rejected until $C$ fills the deficit. If $S$ does not give receipts, $C$ logs the event and reports to $A$ after a threshold number of occurrence. In cases where the providers use load balancing, the client may not know the destination servers, so the client will not be able to generate receipts. One way to solve this problem is to ask the server to fetch the receipts periodically rather than asking the client to push them to the server.

## 4.2. Security Analysis

In this section, we briefly analyze the security of the protocols described in section 3.4.3 and explain why $A$ colluding with all clients of SSC cannot deny their consumption, and why $B$ colluding with all of its servers cannot inflate SSC's usage. During the accounting phase, since $B$ has to present to $A$ all of the receipts given by $A$'s clients, the only way $B$ (or its servers) can inflate usage (generate more receipts) is by forging some of the receipts. In order to forge receipts, $B$ has to either compute the pre-image of the last receipt, or forge a client's signature. Since clients are using a cryptographically strong hash function and a secure signature scheme (for signing the root), forging receipts is computationally infeasible.

Each receipt chain is signed by the client and is unique per client-server pair. Therefore, assuming a strong signature scheme (e.g., RSA), the only way the server would have received a receipt is from the client. Therefore, $C$ and $A$ cannot deny their usage. If $C$ fails to give receipts, then $S$ can easily detect that and take appropriate actions according the local policies. Similarly, since each chain is committed by a client to one specific server, servers cannot claim multiple credits for the same receipt by forwarding the receipt to other servers. Following the same arguments for deletes, it is straightforward to see that clients cannot forge delete receipts and nor can the servers deny their receipts.

## 4.3. Performance Analysis

Let $n$ and $m$ indicate the number of clients and the number of servers in the system. Let each client create $k$ bytes of additional storage, consume $l$ bytes of bandwidth, and delete $d$ bytes of data per server in accounting period $t$. Let chunk sizes be equal to $b$. Therefore, for each server a client has to give $\frac{k}{b}$ `storage_receipts` and $\frac{l}{b}$ `bandwidth_receipts`. For the rest of this section, let us also assume that $A$ has to verify all of the receipts everyday, that is, we calculate the *worst case* computation cost on $A$, *to resolve a dispute*.

**Without Deletes** Let us first ignore `delete_storage_receipts` and compute the overhead for accounting of storage and bandwidth usage. At the beginning of each accounting period, each client has to compute two hash chains per server of lengths $k/b$ and $l/b$ respectively. The total cost of hash computation on the client per period $t$ is $\frac{k+l}{b} * m$ and the cost to sign each root is $2 * m$ signature generation. Assuming that the hash chains are pre-computed at the beginning of every accounting period, during the data path the client does not have to perform any computation. Whereas, during data path, each server performs one hash computation (to verify receipt) per data transfer of size $b$. Thus, over the accounting period $t$, each server has to perform $\frac{k+l}{b} * n$ hash operations and verify $2*n$ signatures (for commitment certificates). To resolve a dispute, $A$ has to verify all of the receipts and signatures generated by its clients. Thus, in the worst case, $A$ performs $\frac{k+l}{b} * m * n$ hash operations and $2 * m * n$ signature verifications.

Now let us compute the storage and cache cost at each entity. At the beginning of the accounting period, $C$ has to cache a pair of hash chains for each server, which is $\frac{k+l}{b}$ hash values. It is important to note that this is the worst case cost; once a hash value is used $C$ can delete the receipt from its cache; thus, the cache space decreases over time. In addition, there are several ways of reducing the cache space [14, 27]. Since generating hash chains is inexpensive, $C$ can tradeoff space with computation by caching only some intermediate values and compute the values in between whenever necessary by utilizing the idle CPU while performing I/O. Each server has to cache only the last verified receipt from every client. Hence, the total number of receipts cached by each server is $2*n$ receipts. Since $A$ and $B$ have to store only the last receipts received from each client, the storage cost on them is $2 * n * m$ receipts and $2*m*n$ signatures. Additional bandwidth requirement between every client and server is equal to the number of receipts sent by the client to the server, which is $\frac{k+l}{b}$ receipts. Bandwidth overhead between each $S$ and $B$ is $2*n$ receipts $+ 2 * n$ signatures. Total bandwidth overhead between $A$ and $B$ is $2 * n * m$ receipts $+ 2 * n * m$ signatures.

**With Deletes** Each server has to generate $\frac{d}{b} * n$ receipts and perform $n$ signature operations to sign the commitment certificates of each chain. Each client has to perform one hash operation per deleted chunk. The computation overhead on $A$ due to deletes is $\frac{d}{b} * n * m$ hash operations and $n * m$ signature verifications. Each client has to cache the latest receipt from each server, thus, a total cache space of $m$ receipts. The worst case cache space on each server is of $\frac{d}{b} * n$ receipts. Bandwidth overhead between each $S$ and $B$ is $n$ receipts, since every server has to give only the last used receipt. Since $B$ forwards all receipts received form its servers, the bandwidth overhead between $A$ and $B$ is $n * m$ receipts. Finally, bandwidth overhead between each $C$ and $A$ is $m$ receipts.

**Example** Let us take an example to understand the protocol overhead. Here, we mainly focus on the cost due to hash computations as it is the most dominant cost. Let us assume that the number of clients $n$ = $2^{10}$ and number of servers $m$ = $2^3$. Let the time interval $t$ be one day and chunk size $b$ = 4KB. Let the consumer organization read 3TB/day and let the increase in storage space be 1TB/day. Thus, the total bandwidth consumed is 4TB/day. Note, these are quite generous numbers. Storage in big medical companies such as Mayo clinic grow at a rate of 1TB per nine days [28]. A SSC will be typically of smaller scale[2].

The amount of bandwidth ($l$) and storage space ($k$) consumed by each client per server is $4 * 2^{27}$ and $2^{27}$ bytes respectively. Therefore, total computation cost to generate `bandwidth_receipts` and `storage_receipts` on each client = $2^{20}$ hash operations. Using OpenSSL [23] 0.9.8, an average 3 GHz Pentium 4 can perform approximately $2^{20}$ hashes in a second. This will take only 1 second of CPU time. On a typical client machine, to pre-generate receipts, stealing one second of idle CPU time from an entire day is not difficult. Every server has to perform one hash operations per chunk, which will incur $1\mu$s delay during datapath. The worst case computation cost per day on $A$ will be $2^{30}$ ($\frac{(4*2^{27}+2^{27})}{2^{12}} * 2^{10} * 2^3$) hash computations, which will

[2]Unfortunately, we could not find any work on SSC usage trends.

take $1000$ seconds or less than $17$ minutes. Considering the scale of usage, this is a small cost on an accounting server for 5 TB of usage consumption. Thus, the accounting server does not have to be powerful machine. It is clear from this example that the computation overhead imposed by Saksha during data path as well as on $A$ will be small. Assuming SHA-256 the worst case cost on each client to cache the two pair of chains is $2^{20} * 2^5$ = 32 MB, which is less than $0.8\%$ of the amount of bandwidth and storage consumed by each client. Total bandwidth overhead between $A$ and $B$ is approximately 2MB for $2 * n * m$ 1024-bit RSA signatures and $0.6MB$ for $2 * n * m$ hash values.

Let us assume that each client deletes $d$ = $1MB$ from each server. That is, total data deleted is $8GB$ per day ($1GB$ per server). Then, each server has to generate $2^{10}$ hash chains of length $2^8$. Thus, total computation cost per day on each server is a little less than one second. Total worst case cache cost (using SHA-256) on each server is $8MB$. Additional computation cost on $A$ due to deletes is only 4 seconds.

## 5. Implementation Details

Figure 2 shows the three user-space components that comprise Saksha: the Saksha accounting client layer ($\texttt{saksha}_C$), the Saksha accounting server layer ($\texttt{saksha}_S$), and the receipt server (rcpt_srv). The Saksha accounting client is layered on the top of CoreFS [2], a simple user-space networked file system that is designed to be mounted through an interface provided by FUSE [9]. The Saksha accounting server $\texttt{saksha}_S$ is layered on top of the CoreFS file server, which is a user-space daemon that accepts clients' file system requests on a TCP socket. We chose FUSE because it allowed us to implement our concepts in user space without having to manipulate complex kernel code. We used CoreFS because it is already a networked file systems and simple enough to be used for rapid prototyping. When users access files, the FUSE kernel library directs the file system requests to the CoreFS client, which forwards these requests to $\texttt{saksha}_C$. $\texttt{saksha}_C$ creates an accounting message, inserts the file system request in this message without any modification, piggybacks receipts (if any), and sends the accounting message to $\texttt{saksha}_S$. $\texttt{saksha}_S$ verifies the receipts given by the client and passes the file system request to the CoreFS server. The sequence of operations is reversed for the response message. The rcpt_srv program is portable to any Unix-like operating system. The purpose of the rcpt_srv is to pre-generate receipts (hash chains) and provide them to the accounting layer when ever they demand. It could easily monitor the number of chains consumed by the accounting layer and generate more receipts if required. The rcpt_srv program can also be used for other systems that require generating receipts as well, for example, micropayments,or measuring web clicks. All cryptographic operations are performed using OpenSSL 0.9.8 [23].

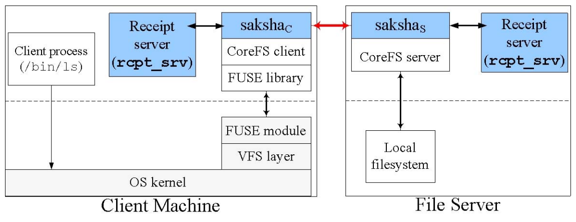
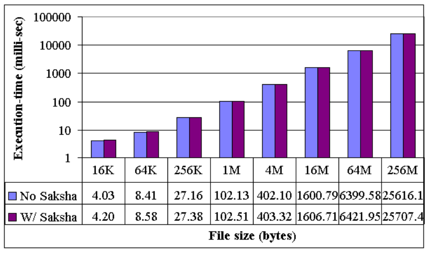


**Figure 2. The Saksha system components.**

**Receipt generation and exchange** Every client and server machine runs one rcpt_srv in the background to get storage/bandwidth receipts and delete receipts, respectively. The rcpt_srv communicates with the accounting layer over Unix domain sockets. When the CoreFS client establishes a connection with a file server $S$, it informs $\texttt{saksha}_C$ about the newly established connection, which in turn requests the receipt server to create and return commitment certificates (signed roots) for storage and bandwidth chains for $S$ as well as the receipts (hash values) of each chain. The rcpt_srv picks two of the pre-generated hash chains (or creates new chains if necessary), creates the commitment certificates (as described in section 3.4.3), and returns the commitment certificates and receipts to $\texttt{saksha}_C$. After which, $\texttt{saksha}_C$ sends the two commitment certificates to $S$. In response, $\texttt{saksha}_S$ acquires the commitment certificates and receipts for the delete chain, and sends the commitment certificate to $\texttt{saksha}_C$. If the chains expire (due to change in the accounting interval) or get used up during an ongoing connection, the commitment certificates are re-acquired and re-exchanged.

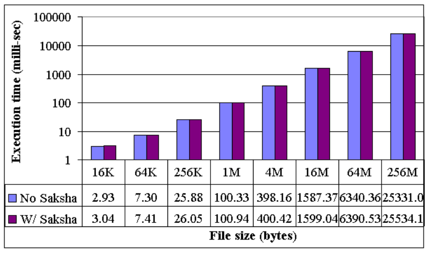
During normal file system operations, the accounting layers measure the amount of storage increased, storage decreased, and bandwidth consumed due to the various file system requests. $\texttt{saksha}_C$ and $\texttt{saksha}_S$ layers piggyback the necessary receipts to file system requests and responses, respectively. Piggybacking the receipts avoids unnecessary addition of explicit network rounds and the resulting latency overhead. The accounting layers store the commitment certificates and the $\langle index, value \rangle$ pair of the last receipt in a local file. These receipts can be transferred for accounting to the accounting servers through mechanisms external to Saksha. The Saksha accounting program (not shown in the figure) can read these receipts and perform accounting as explained in the usage measurement phase in section 3.4.3.

**Modularity** One of the main goals while implementing $\texttt{saksha}_C$ and $\texttt{saksha}_S$ was to implement them in a way that will require minimal changes to the underlying file system layer. Our goal was to keep the accounting logic sepa-



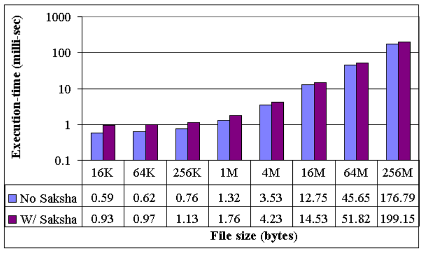
| | 16K | 64K | 256K | 1M | 4M | 16M | 64M | 256M |
|---|---|---|---|---|---|---|---|---|
| ▨ No Saksha | 4.03 | 8.41 | 27.16 | 102.13 | 402.10 | 1600.79 | 6399.58 | 25616.1 |
| ■ W/ Saksha | 4.20 | 8.58 | 27.38 | 102.51 | 403.32 | 1606.71 | 6421.95 | 25707.4 |

write



| | 16K | 64K | 256K | 1M | 4M | 16M | 64M | 256M |
|---|---|---|---|---|---|---|---|---|
| ▨ No Saksha | 2.93 | 7.30 | 25.88 | 100.33 | 398.16 | 1587.37 | 6340.36 | 25331.0 |
| ■ W/ Saksha | 3.04 | 7.41 | 26.05 | 100.94 | 400.42 | 1599.04 | 6390.53 | 25534.1 |

read



| | 16K | 64K | 256K | 1M | 4M | 16M | 64M | 256M |
|---|---|---|---|---|---|---|---|---|
| ▨ No Saksha | 0.59 | 0.62 | 0.76 | 1.32 | 3.53 | 12.75 | 45.65 | 176.79 |
| ■ W/ Saksha | 0.93 | 0.97 | 1.13 | 1.76 | 4.23 | 14.53 | 51.82 | 199.15 |

unlink

**Figure 3. Wall clock execution time of different phases of the experiment.**

rate from the file system and to make the accounting layers modular so that they can be easily integrated with CoreFS and potentially other file systems. To achieve this, Saksha defines simple APIs that the file system layer has to call from appropriate locations. We illustrate this with an example of accounting for a increase in storage space, which primarily involves calling three Saksha functions, `account_init`, `account_open`, and `account_write`. When the user mounts the CoreFS file system, the CoreFS layer calls the `account_init` function and passes the pointer to its `getattr` function that gets file size and other status information. The `account_init` function records this function pointer. When the CoreFS client receives an `open` request (to open a file), the client invokes the `account_open` function, which in turn invokes the `getattr` function of the underlying file system, acquires size information of the client and caches it. Finally, when the CoreFS client successfully completes a `write` request, it calls `account_write` function, which based on the file size, offset, and current write size determines whether the client has acquired additional storage space (as discussed in section 3.2) and takes appropriate actions. Similarly, for bandwidth measurement, the CoreFS layers have to invoke the appropriate higher-level functions. Thus, accounting for a write operation took adding only five lines of code to the CoreFS layer.

One way to implement piggybacking was to modify the file system layers in the following manner. For each outgoing file system message the file system layer should query the accounting layer for receipts, and piggyback the receipts to the outgoing message. The receiver will perform the reverse, remove the piggybacked receipt from the file system message and send it to the accounting layer for verification. However, because this approach requires changing the file system message structures (and networking functions), it will require significant changes in the file system layers. Further, these complex modifications will have to be performed for every file system using Saksha. To reduce the complexity of these modifications, Saksha has defined APIs, which when invoked by the file system layer, allows Saksha to trap the outgoing file system requests. As a part of the `account_init`, the file system layer passes a `context` structure, which among other fields, contains pointers to `send` and `receive` functions. In `account_init`, the Saksha layer assigns the address of its own send and receive functions to the respective pointer. When the file system layer is about to send a message over the network, instead of invoking its send function, it invokes the Saksha's send function by dereferencing the `context`'s send function. The Saksha `send` function considers the file system message as an opaque object, piggybacks the receipts, and sends the encapsulated accounting message to the receiver. At the receiver's end, the reverse operation is performed; the file system layer invokes the Saksha's receive function, which after receiving the accounting message, verifies the receipts and extracts the opaque file system message and returns it to the file system layer. While this approach also requires some modifications at the file system layer, it keeps these modifications minimal. Integrating support for piggybacking into CoreFS required changing approximately forty lines of code, of which most of the changes had to be performed to modify the function prototypes to pass the `context` structure to the necessary functions.

## 6. Performance Evaluation

In section 4.3 we discussed the absolute CPU, storage, and bandwidth overhead on each of the system entities. In this section we measure the overhead in terms of latency during the file system read, write, and unlink operations. The goal of this experiment is to measure for different file sizes the amount of extra latency that will be visible by Saksha clients because of its accounting mechanisms. We measured the file system performance for each of these operations between two 3GHz Pentium IV machines with 1GB of RAM, 100 Mbit/sec Ethernet link, and running SUSE 10.1 with Linux 2.6.13. The file server was run on top of Reiserfs 3.6.19, the default SUSE 10.1 file system.

By default FUSE sets the maximum size of each write operation to 4KB, which will result in many network rounds while writing a large file to the server and hide the overhead of hash verifications. Therefore, FUSE was run in direct IO mode, which allows us to set the size of a write operation to be greater than 4KB. Neither the CoreFS client nor the CoreFS server performed any caching. The experiment was performed in three stages: the first stage creates a new file on the file server and then sequentially writes to the file with maximum size of each write operation set to 64KB, the second stage sequentially reads the file from the file server, and the third stage deletes the file using unlink operation. We measured the time required for completion of each of the phases with and without Saksha layered on the top of CoreFS. The above experiment was performed for different file sizes with a constant chunk size of 16KB. Figure 3 shows the execution time of each phase in milliseconds on a log-scale and the actual values (mean of fifteen runs) in the tabular format.

For each phase, the overhead is higher for small file sizes and reduces with the increase in the file size. This is because as the file sizes increase the network and disk latency dominates the computational latency. Clearly, for the read and write phases, the performance of Saksha is comparable to that of CoreFS. For small file size of 16KB the overhead for read and write was 3% (only $110\mu$s) and 4% (only $170\mu$s), respectively. The overhead for both the phases quickly drops below 1% after 64KB file size. Because unlink is a simple operation that modifies only the meta-data of the file system and is not as IO intensive as read and write operations, the overhead of accounting mechanisms is more visible for the unlink operation (although the absolute value of the overhead is low). For 16KB file size, the overhead is around $340\mu$s (58%) and the overhead drops below 20% after 4MB file size and reaches 13% for 256MB file size. We would like to stress that since unlink operations are infrequent (compared to read/write), incurring slightly more overhead during the unlink operations will not significantly affect the performance of the system.

## 7. Related Work

The closest work to our research is by Gentry et. al [10] and by Ioannidis et al. [13]. The goal of Gentry et. al was to account for clients' phone usage, especially for roaming calls so that the foreign service provider can charge the user's home service provider for the phone service used by the user. They proposed to use QuasiModo trees for accounting. QuasiModo trees require more cache space on the verifier, which can be high in the case of storage servers, e.g., $2GB$ per server using example in section 6. While using QuasiModo trees will reduce the computation cost on $A$, this advantage is not that significant in practice since $A$ can be dedicated machine (or a machine idle for few minutes per day). In addition, various techniques [14, 27, 12] can be used to further reduce the computation cost (of our scheme) due to hash chains. Besides, in our protocol, hash verifications have to be performed only to resolve disputes. Nevertheless, one can certainly use QuasiModo tree instead of hash chains or a combination of hash chains and QuasiModo tree (as also described in [10]). Filetellar [13] is a credential based network storage system where a user stores files on the server and pays using KeyNote microchecks. A user can delegate rights to another user and can get paid if the delegatee access his files. The focus of this paper is somewhat different from ours. Filetellar is more suitable for small-scale web-based file sharing, such as sharing of photos, where the number of users accessing the files is small, the number of transactions with the storage servers is small, and the data transferred during each transaction is large. It is not clear how their system handles overwrites and updates of data and meta-data as well as the bandwidth consumption resulting from file system operations. Each transaction (microcheck) requires a signature, which is expensive for file systems where there can be a large number of storage transactions and each transaction can be of a small size. Micropayments (some of which could be used instead of hash chains) can be found in [20].

## 8. Conclusions

Establishing trust between consumers and providers is key to the success of storage outsourcing. We believe that by including security as a part of its service, a storage service provider will be able to gain trust from its consumers. In this paper, we have presented Saksha, a system that provides a light-weight solution to one of the security challenges - automated and secure accounting. By including Saksha, providers will be able to give verifiable proof of usage to its consumers. Saksha prevents providers from inflating its consumers usage values, and consumers from denying the usage. To the best of our knowledge, Saksha is the first accounting system that can be layered on the top of networked file systems. Because of its modular architecture, Saksha decouples secure accounting from the underlying file system and keeps modifications to the file system layer minimal. Our preliminary performance evaluation demonstrates that Saksha incurs small overhead during file access and can be deployed in practice.

**Acknowledgments** This research was supported in part by the National Science Foundation (NSF) under grants CNS-0448423 and CCF-0621462, and by the DTC Intelligent Storage Consortium (DISC), University of Minnesota. We thank Anjali Joshi, Peng Wang, and the anonymous reviewers for their valuable comments and suggestions.

## References

[1] Amazon S3-simple storage service. `http://aws.amazon.com/s3`.

[2] CoreFS: A basic networked file system. http://www.cs.umn.edu/research/sclab/coreFS.html.

[3] HP Managed Storage Solution. http://h71028.www7.hp.com/ERC/downloads/5982-3551EN.pdf.

[4] WorldCom faces criminal charges. CBS News. http://www.cbsnews.com/stories/2003/08/03/national/main566401.shtml.

[5] R. Anderson, H. Manifavas, and C. Sutherland. A practical electronic cash system. Manuscript, 1995.

[6] C. Blundo and S. Cimato. A software infrastructure for authenticated web metering. *Computer*, 37(4), 2004.

[7] W. Chai. EMC unveils pay-per-use storage. http://news.zdnet.co.uk/business/0,39020645,2137616,00.htm.

[8] A. Freedman. Storage Service Please: SSPs Start to Make Progress, Again. IDC, February 2005. http://www.gdv.ca/files/IDC_Feb2005.pdf.

[9] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.

[10] C. Gentry and Z. Ramzan. Microcredits for verifiable foreign service provider metering. In *Proceedings of Financial Cryptography and Data Security*, pages 9–23, 2004.

[11] R. Hasan, W. Yurcik, and S. Myagmar. The evolution of storage service providers: techniques and challenges to outsourcing storage. In *ACM StorageSS*, pages 1–8, 2005.

[12] Y.-C. Hu, M. Jakobsson, and A. Perrig. Efficient constructions for one-way hash chains. In *Proceedings of Applied Cryptography and Network Security*, pages 423–441, 2005.

[13] J. Ioannidis, S. Ioannidis, A. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *Proceedings of Financial Cryptography*, pages 282–299, 2002.

[14] M. Jakobsson. Fractal hash sequence representation and traversal. In *Proceedings of IEEE International Symposium on Information Theory (ISIT)*, 2002.

[15] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus-scalable secure file sharing on untrusted storage. In *USENIX File and Storage Technologies*, 2003.

[16] V. Kher and Y. Kim. Securing distributed storage: challenges, techniques, and systems. In *ACM StorageSS*, 2005.

[17] L. Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI, 1979.

[18] L. Lamport. Password authentication with insecure communication. *Commun. ACM*, Nov 1981.

[19] J. Li, M. Krohn, D. Mazires, and D. Shasha. Secure untrusted data repository (SUNDR). In *USENIX OSDI*, pages 121–136, December 2004.

[20] R. J. Lipton and R. Ostrovsky. Micropayments via efficient coin-flipping. In *Proc. of Financial Cryptography*, 1998.

[21] E. Miller, D. Long, W. Freeman, and B. Reed. Strong security for distributed file systems. In *Proceedings of USENIX File and Storage Technologies (FAST)*, January 2002.

[22] A. Odlyzko. Internet pricing and the history of communications. *Computer Networks*, 2001.

[23] OpenSSL 0.9.8. http://www.openssl.org/.

[24] T. P. Pedersen. Electronic payments of small amounts. In *Security Protocols Workshop*, pages 59–68, 1996.

[25] R. L. Rivest and A. Shamir. Payword and micromint: Two simple micropayment schemes. In *Security Protocols Workshop*, pages 69–87, 1996.

[26] T. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the IEEE International Conference on distributed Computing Systems (ICDCS)*, 2006.

[27] Y. Sella. The computation-storage trade-offs of hash chain traversal. In *Financial Cryptography*, pages 270–285, 2003.

[28] N. Spillers. Storage challenges in the medical environment. www.dtc.umn.edu/disc/isw/presentations/isw4_11.pdf.

[29] S. Woolley. Baby Bell accounting fraud uncovered. Forbes News. http://www.forbes.com/forbes/2003/0512/082.html.

[30] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.