Secure Locking For Untrusted Clouds

Chiu C. Tan[†], Qin Liu[†‡], and Jie Wu[†]
[†]*Department of Computer and Information Sciences, Temple University, USA*
[‡]*School of Information Science and Engineering, Central South University, P. R. China*
Email:{cctan,jiewu}@temple.edu, gracelq628@yahoo.com.cn

***Abstract*—Migrating applications with strong consistency requirements to public cloud platforms remains risky since the *data owner* cannot verify the correctness of the public cloud's locking algorithm. In this paper, we identify new attacks that an untrusted *cloud provider* can launch via control of the locking mechanism and propose an extension to existing locking scheme to address such attacks. Our solution modifies the read and write locks to include a short history to allow *data users* to verify the correctness of their assigned locks, and can also prevent the cloud from re-ordering operations for financial gain.**

***Keywords*-Cloud computing; security; locks; read/write access**

I. Introduction

Cloud computing has drawn much attention in recent years. The ability for users to dynamically scale their IT operations depending on their needs, without making expensive upfront investments, is a key attraction of using commercial cloud computing services. Examples of applications using such cloud services including using the cloud to perform data mining [1] and other applications that archive digital collections [2]. Many business and government entities have indicated interest in increasing their use of cloud services [3].

We believe that the next wave of cloud computing innovations will shift away from cheaper storage and faster response times towards offering higher value services, such as concurrency control. Such offerings will allow data owners to outsource even more IT operations, and cloud providers can use this as a means of distinguishing themselves from the competition. However, since third party cloud providers are considered untrustworthy [4]–[6], we need to consider the security implications of an untrusted cloud providing such services. Here, an untrusted cloud will not perform malicious actions, such as returning incorrect data to a user, but may try to learn additional information from the stored data, or subtly manipulate the protocols for financial benefit. We also describe an untrusted cloud as a *misbehaving cloud* in this paper. We will further explore the actions of misbehaving cloud in subsequent sections.

In Fig. 1(a), we see that prior to outsourcing data to the cloud, the data owner has to maintain a larger data storage system, and data users will access the data from the data owner directly. The data owner can implement traditional distributed system concurrency control techniques, such as locks, to ensure consistency. Thus, when Alice is updating the sensitive data, the system can issue Alice an exclusive lock which will prevent Bob from reading that data. After Alice has completed her updates, she releases the lock, and Bob can then read the data.

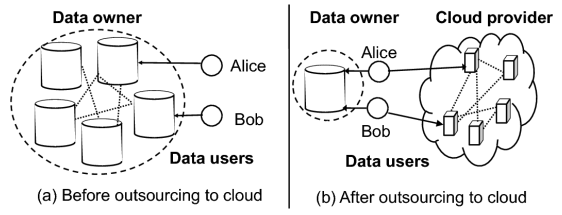

Figure 1. Overview of data owners and users accessing a public cloud.

Fig. 1(b) depicts the setup after the data owner has migrated his data to a cloud provider. We see that the owner needs only to maintain a small system to authenticate users, while the cloud manages the data. To prevent the untrusted cloud from learning the contents of the stored data, one approach is to allow the data owner to store only encrypted data on the cloud, and let users wanting to access data to first obtain permission from the data owner and receive the necessary keys to decrypt the data. The users can then obtain the data from the cloud and decrypt it to receive the information [7]. A misbehaving cloud does not learn the data contents since all the information is encrypted. In essence, after migrating the system to a cloud, the data owner can still control data access through the encryption, while leaving the cloud to be responsible for data storage operations.

However, since the data owner does not have control over the system, the owner now has to rely on the untrusted cloud provider to execute the concurrency control algorithm. The untrusted cloud can attempt to manipulate the locking algorithm to derive additional benefits. For example, consider a cloud provider that is required to meet certain user query response times. (Such provisions are increasingly popular in cloud deployments to ensure quality-of-service (QoS).) Now, let us assume that the cloud receives a string of requests R, W, R, W, R, W within a period of time, where R is a read request, and W is a write request to the same data. To complete this sequence of operations correctly, the cloud will have to lock after every access, and unlock once done, leading to a total of 6 lock operations. On the other hand, the untrusted cloud can re-order it to R, R, R, W, W, W, and only need to perform lock operations 4 times. Any read request after the first read is essentially free, since locks can allow concurrent reads, and the cloud can give immediate access without incurring any resources. A misbehaving cloud that has too many clients and cannot meet the QoS requirements will re-order the operations to avoid paying the penalty. Traditional concurrency control research does not address this type of problem since it assumes that the system running the algorithm is trusted, while existing current cloud security research focusing mainly on improving operations on encrypted data [8] cannot be applied to this problem.

In this paper, we propose CloudLock, an efficient locking protocol for an untrusted cloud providing infrastructure as a service (IaaS). Our key contributions are as follows:

1) We are the first to explore possible attacks that can be launched by an untrusted public cloud service provider implementing concurrency protection using locks.
2) We extend existing locking algorithms to be implemented by an untrusted cloud service provider. The modification of well-studied locking techniques make our solution more robust.
3) Our protocols modify the read and write locks to allow users to quickly detect locking violations to ensure the correctness of the data, and can also be used as evidence to penalize the cloud provider.

We stress that the goal of this paper is to *compliment* existing locking schemes to allow them to be executed by an untrusted cloud provider, and *not* to design new schemes. The rest of our paper is as follows: We present the problem formulation in Section II, and our solution in Section III. Section IV contains some additional discussion, Section V contains related work, and Section VI concludes the paper.

II. Problem Formulation

We consider a *data owner* wanting to store data in a public cloud for multiple *users* to access. As shown in Fig. 1, all users have to receive permissions to access the data from the data owner.

A. System requirements and setup

The system requirements are expressed as a service level agreement (SLA) between the owner and the cloud. There are three requirements. The first is the *confidentiality* requirement. This means that only authorized users can obtain the data. The cloud provider is not considered an authorized user. For a small portion of sensitive data, the owner requires that the cloud provide a *consistency* requirement. This means that any valid user will never read stale data, and any data updates once committed will be read by any subsequent read. Finally, the last requirement is a *latency* requirement, where the cloud has to satisfy a user's request within a given time.

We assume that the data owner maintains a server for the purposes of authorizing users, issuing commands to the cloud, and logging any information from authorized users to keep track of the latency requirement. There is also a central time server that can be maintained by the owner or by a neutral third party. Before migrating the data to the cloud, the data owner will generate a pair of public and private keys. The data owner will also encrypt each piece of data with its own data encryption key before uploading it to the cloud provider.

A user, after being authorized by the data owner, will be issued a pair of public and private keys by the data owner, as well as the data encryption and decryption keys for each piece of data he is authorized to access. With these keys, the user can decrypt the data obtained from the cloud provider.

The cloud provider will run our locking scheme, which will ensure that a user's updates to the data will be reflected by all subsequent users accessing it. Since locking is an expensive operation (a write lock, for instance, precludes any other user access to the data), locks will only apply to sensitive data, as defined by the data owner. The cloud provider will have a pair of public and private keys that is used to create digital signatures.

We will first assume that any user will be able to verify any other valid user's signature, and elaborate more on this later. We also assume that all communications between the owner, cloud, and users are performed over secure channels and are free from eavesdropping. Table I indicates the notation used in this paper.

PK_u,SK_u	User public/private keys
PK_{cloud},SK_{cloud}	Cloud public/private keys
rln_i, wln_k	Read (write) lock number i (k)
$S(d, k)$	Signing data d with key k.
$V(s, k)$	Verify signature s with key k.
$E(d, k)$	Encrypt data d with key k.
$D(c, k)$	Decrypt ciphertext c with key k.

Table I
Table of notation

B. Adversary model

This paper is primarily concerned with defending against a misbehaving cloud provider. We use the term "misbehaving" instead of "malicious"' because the cloud will generally adhere to the protocols, but may decide to cheat for financial benefits. This is stronger than the "honest-but-curious" assumption usually made in cloud security research, since the cloud will break protocol if there is a financial benefit in doing so.

We assume the misbehaving cloud can launch *confidentiality attacks* by trying to learn more about the contents of the cloud data. This is a common type of attack considered in cloud computing security [7], and is commonly addressed by storing encrypted data in the cloud. Our solution also adopts this same principal. A misbehaving cloud is capable of launching *locking attacks* to manipulate the locking algorithm to achieve economic gains. In this paper, we identify a lock by its lock number. So, a lock with lock number i should be issued after lock number i-1, and so on. The types of locking attacks are as follows:

1) Issue incorrect lock number.
2) Issue the same lock to two or more users.
3) Fraudulently claim that the lock busy.
4) Deny issuing a lock.
5) Re-order user requests before issuing a lock.

The cloud's motivation behind the locking attacks is economic rather than malice. Providing efficient service with concurrency protections involves more powerful hardware, higher energy costs, and more network resources, all of which incurs a higher operation cost for the cloud provider. For example, an incorrect lock number may be issued because the cloud provider did not devote sufficient resources to ensure changes are propagated to all the replicas, or that the cloud may pretend that a lock is unavailable to mask latency delays. The cloud may also schedule the order of operations to minimize locking operations. This could have a negative impact on users by delaying important updates.

Finally, our adversary model assumes that there is no collusion between the different parties. Therefore, the cloud and users will not collude to cheat the owner, nor will the owner and users collude to frame the cloud provider. We also assume that all users are honest and will not launch attacks, such as not releasing the lock or to fraudulently claim to possess a lock.

III. CloudLock Protocol

CloudLock allows multiple users to read the data concurrently since the data remains consistent. Only when a user wants to write to the data do we need to exclude other users. We adopt the rule from [9], where a read lock is considered a *shared* mode lock, and a write lock is considered an *exclusive* mode lock. Once a data object is locked in an exclusive mode, no other user can lock that object in either mode. A data object locked in a shared mode can allow access to other users.

A. Overview

The CloudLock protocol consists of interactions between four parties: the data owner, a central time server, a data user, and the cloud provider. The overview of operations is that each time a user wishes to access data, the user will first obtain a timestamp from the central time server before querying the cloud for data. The purpose is to detect any re-ordering of operations from the cloud. We assume that there is a bounded delay Δt. The cloud, after waiting for Δt, will have all the pending requests in a buffer. This process is illustrated in Fig. 2. We see that the cloud will re-order the

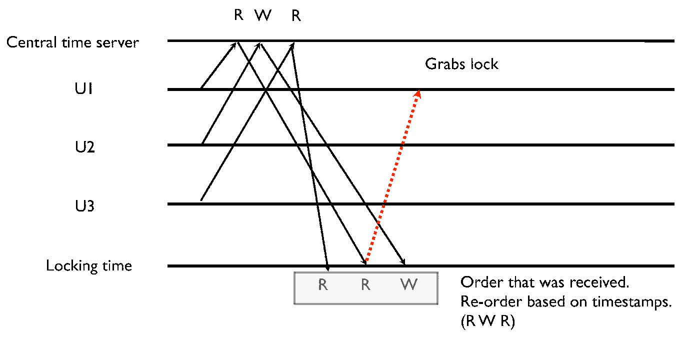

Figure 2. Reordering using a timestamp

request based on the timestamp value from the central time server.

After receiving the user's request, the cloud will then perform the necessary locking operations and return the data and some other verification information to the user (we will elaborate on this later). When the user releases the lock and data updates, if any, back to the cloud, the user will also update some information with the data owner, which maintains a table, shown in Table II. Using this table, the data owner is able to determine when a lock number was requested, issued, and released, and thus determine whether the latency requirement was met by the cloud.

Lock number	Requested	Received	Returned
...	...	...	...

Table II
TABLE MAINTAINED BY DATA OWNER.

B. Protocol description

CloudLock uses two locks, a read lock and a write lock, denoted as $RLOCK(i)$ and $WLOCK(k)$, where $RLOCK(i) = rln_i, S(rln_i, SK_{cloud})$, and $WLOCK(k) = wln_k, S(wln_k, SK_{cloud})$. The read lock number and write lock number are rln_i and wln_k. We use "i" to denote a generic read lock number, and "k" to denote a generic write lock number, for clarity.

Our solution relies on maintaining two sets of lock numbers, each time the cloud returning both a read **and** a write lock number to the user. The intuition is that a user holding a $RLOCK$ will use the $WLOCK$ to verify that he is indeed reading the latest version, and that a user holding a $WLOCK$ will use the $RLOCK$ to determine whether he has been waiting too long.

CloudLock uses the following rules to ensure safety: First, multiple read locks can be issued safely; second, a write lock can only be issued when all read locks are released; third, while a write lock has been issued and not yet recovered, no other locks, read or write, can be issued. Table III shows the four possible cases, and Fig.. 3 shows the interactions between the user and the cloud for each of the four cases.

	Request read	Request write
Issued $RLOCK$	Case 1	Case 2
Issued $WLOCK$	Case 3	Case 4

Table III
$RLOCK$ AND $WLOCK$ OPTIONS

We use the term *assigned* $RLOCK(i)$ or *assigned* $WLOCK(k)$ as the current read (write) lock number assigned to the user. The term *next* $WLOCK(k)$ indicates that the next write lock number issued will be k. The term *last* $RLOCK(i)$ means that the latest read lock number assigned is i and the next $RLOCK$ number that has not been assigned yet will be $i+1$. When a user releases a lock, he will return a $UNLOCK-RLOCK(i)$ or $UNLOCK-WLOCK(k)$ depending on which lock he requested. The $UNLOCK - RLOCK(i)$ is just the read lock number and accompanying singnature, $rln_i, S(rln_i, SK_{user})$. The unlocking of the write lock is similar.

The term $DATA$ refers to the encryption of the data d, $E(d,K)$, and $DATA'$ refers to the data after being written by a user, $E(d',K)$. The $HIST$ acts like a hash chain linking the current read and write lock numbers with the contents on the data. This operation is performed by a user when he releases a lock. A $HIST(k,i)$ is thus $\{H(wln_k|DATA)$, $S(H(wln_k|DATA), SK_{user})$, rln_i, $S(wln_k|rln_i, SK)user)\}$. For brevity, we will omit the signatures $S(H(wln_k|DATA), SK_{user})$ and $S(wln_k|rln_i, SK)user)$ in future discussion. Details of the four cases are given below:

Case 1: This case occurs when a read lock as already been issued and another user also wants a read lock. In this case, we want to allow this user to read immediately. As shown in Fig. 3(a), in addition to the lock numbers and $DATA$, the cloud will return $HIST(k-1,<i)$. The $<i$ refers to a read lock number that is smaller than i. This $HIST(k-1,<i)$ is just $H(wln_{k-1}|DATA)$, and its accompanying signature, and $rln_{<i}$, $S(wl_{k-1}|rln_{<i})$. The user will use $WLOCK(k)$ to verify that $H(wln_{k-1}|DATA)$ to ensure that he is reading the latest $DATA$. Then, the user checks that his read lock rln_i is correct. There could be other users who have been issued rlns before-hand, but in any case, their values, $rl_{<i}$, have to be less than rl_i. The user also checks that the smaller rln value is correctly associated with wln_{k-1} by checking the signature $S(wln_{k-1}|rln_{<i})$. The user can only check $rl_{<i}$, and not strictly rl_{i-1}, because the last read lock issue could be the first read lock returned.

Case 2: This case occurs when a user wants to do a write, but there is one or more read locks being issued. The interactions are shown in Fig 3(b). The user will first receive the assigned $WLOCK(k)$ and the last $RLOCK(i)$. Since there are others holding on the the read lock, the cloud cannot return the data. The user wait until the cloud returns $HIST(k-1,i-1)$ to him. Using $WLOCK(k)$, the user can verify $HIST(k-1)$ to determine that he has been issued the latest $DATA$. The last $RLOCK(i)$ is used check that there is a rln_{i-1} issued to a valid user, preventing the cloud from claiming imaginary users with read locks.

Case 3: This case is where a $WLOCK$ has been issued, and a user wants to read. Using next $WLOCK(k)$, the user can check $HIST(k-1,<i)$ to verify that it is reading the last written copy. The $RLOCK(i)$ is used to check that the smaller $rln_{<i}$ is correctly associated with $S(wln_{k-1}|rln_{<i})$.

Case 4: The final case is where a write lock has already been issued and another user also wants a write lock. As before, the user will first receive the assigned $WLOCK(k)$ and last $RLOCK(i)$. When the cloud eventually returns $DATA$ and $HIST(k-1,i-1)$, the user uses $WLOCK(k)$ to verify $HIST(k-1,i-1)$.

C. Security analysis

Here, we analyze the CloudLock scheme against the various attacks. We first consider the confidentiality attack launched by the cloud. In this attack, the cloud tries to learn additional information about the stored data. We see that in Fig. 3, the stored data is always encrypted with $E(d,k)$, so the cloud does not learn its contents. Similarly, all user updates are also encrypted $E(d',k)$, so the cloud learns nothing about the updates either. Next, we will analyze the locking attacks that can be launched by the cloud.

Locking attack 1: *Issuing an incorrect lock number.* This attack arises from an error in the cloud's locking system, possibly due to devoting insufficient resources needed to reach all the replicas.

Each time the user is given the $HIST$, the user can check whether the read lock number rln_i and write lock number wln_k when hashed, correctly matches the values in $HIST$. The cloud cannot attempt to generate a fake $HIST$ because each $HIST$ needs to be signed by a valid user with knowledge of its private key SK_{user}. Since the cloud does not know this key, the cloud cannot forge a signature.

Locking attack 2: *Issuing the same lock to multiple users.* Under the multilock scheme, we can issue multiple $RLOCKS$. Therefore, this attack consists of two scenarios: one is where a $RLOCK$ and $WLOCK$ are issued at the same time, and the other is when multiple $WLOCK$s are issued. For both cases, this attack can be detected by checking with the data owner's table (Table II).

More specifically, let us consider the first case and assume that Alice has obtained a read lock($RLOCK(7)$, $WLOCK(4)$, $HIST(3,6)$), and before she unlocks, the cloud issues a write lock to Bob ($RLOCK(7)$, $WLOCK(4)$, $HIST(3,6)$). Now, Bob's operation is safe since he is updating the latest copy of the $DATA$. However, Alice is not safe, since the $DATA$ she possess may change before she releases the $RLOCK$. Now, if Alice releases her lock before Bob, her entry will appear in the data owner's

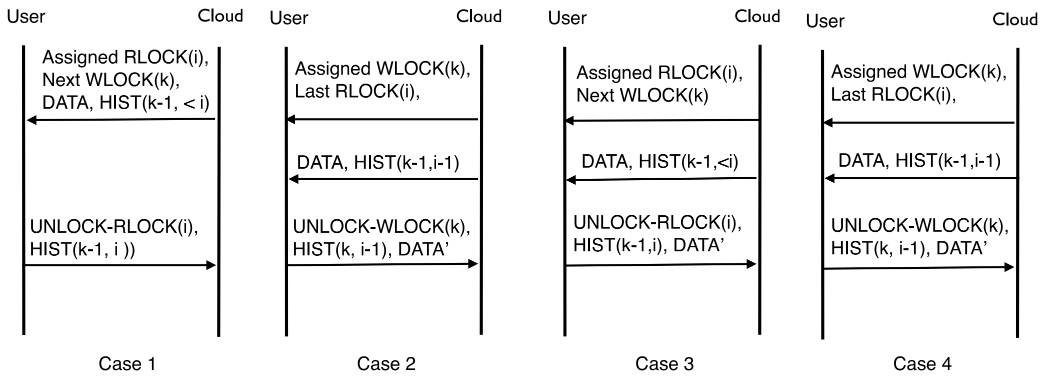
User
Cloud
Assigned RLOCK(i), Next WLOCK(k), DATA, HIST(k-1, < i)
UNLOCK-RLOCK(i), HIST(k-1, i))
Case 1
User
Cloud
Assigned WLOCK(k), Last RLOCK(i),
DATA, HIST(k-1,i-1)
UNLOCK-WLOCK(k), HIST(k, i-1), DATA'
Case 2
User
Cloud
Assigned RLOCK(i), Next WLOCK(k)
DATA, HIST(k-1,<i)
UNLOCK-RLOCK(i), HIST(k-1,i), DATA'
Case 3
User
Cloud
Assigned WLOCK(k), Last RLOCK(i),
DATA, HIST(k-1,i-1)
UNLOCK-WLOCK(k), HIST(k, i-1), DATA'
Case 4

Figure 3. (a)-(d) Cloud user interactions under different cases.

table, and this will be detected when Bob logs his entry to the table, and vice versa. Either way, the attack will be detected and users will be notified. The case of having multiple $WLOCK$s is similar.

Locking attack 3: *Fraudulently claim that lock busy.* The attack occurs when a user wants a $RLOCK$, but the cloud claims that a $WLOCK$ has not been unlocked, or when a user wants a $WLOCK$, but the cloud states that the $RLOCK$ has not been released. In the first instance, lets assume the cloud has to return the next $WLOCK(7)$ number, and later, a supply $HIST(6, < i)$. If the cloud does not supply this $HIST$ value, the cloud will be detected. Since the cloud is bound to return this particular $HIST$, the user can check with the owner's table to determine when $WLOCK(6)$ was issued and unlocked, and detect this attack.

A similar process can be used to detect the second instance. A minor variation is that since multiple $RLOCK$s can be issued concurrently, the cloud could to create phantom $RLOCK$ users. For example, the cloud returns ($WLOCK(3)$, $RLOCK(21)$) to the user wanting the $WLOCK$. In actuality, the read lock number is less than 21. This can be detected as a case of the cloud issuing an incorrect number (locking attack 1), and can be detected when $HIST$ is returned because the cloud is unable to forge the signature for $S(wln_2, rln_2 1)$.

Locking attack 4: *Denying issuing a lock.* The cloud that later detects it has accidentally issued an incorrect lock number and may later deny issuing such a lock. Since all lock numbers have to be signed by the cloud using the cloud's private key, the cloud cannot deny issuing a lock number to a user.

Locking attack 5: *Re-ordering user requests.* This attack can be detected using the central time server. When users append their entries with the data owner after releasing the locks, the timestamp from the central time server can be affixed as well. Re-ordered entries will thus be detected.

D. Key management

Here, we discuss the key management used in CloudLock. The CloudLock protocol requires users to know which public keys to use to authenticate signatures. This can be achieved via a trusted on-line public key directory that can used store a copy of every user's public keys and verify any signatures, or a trusted certificate authority (CA) can be set up to issue certificates, which can be appended to each signature to verify public keys.

In addition, we can also use identity-based cryptography (IBC) in lieu of more traditional public key solution. IBC is a type of asymmetric key cryptography scheme which also makes use of public and private keys [10]. IBC simplifies the problem of distributing and verifying public keys by allowing a user's identity to be used to derive a public key. For example, assuming that Alice's identity is her email address "alice@acme.org", IBC allows users to compute Alice's public key using "alice@acme.org". Using IBE allows us to use novel authentication policies, such as forcing the user to only access the cloud via a certain IP address. This can be accomplished by using the IP address to create the public key.

IV. Additional discussion

A. Alternative design

An alternative method is to avoid using the $HIST$ and instruct the user to record the lock number with the data owner each time he obtains a lock, and also when he releases the lock. In other words, assuming that the data owner is also the central time server, a user needs to do 3 round-trips (one to obtain the timestamp, one when obtaining a lock, and another when unlocking), incurring a cost of 3 round-trip-times (RTTs) to the data owner instead of 2 RTT in our solution. Furthermore, the alternative design also does require stronger synchronization between users.

For example, using the alternative design, Alice is issued $WLOCK(5)$. She records the lock number with the data owner, and checks the table to determine whether $WLOCK(4)$ has been returned. Assuming that $WLOCK(4)$ is missing, it could be possible that either the cloud has issued the wrong lock number (locking attack 1), or that the prior user Bob has not recorded that information with the data owner. Using the hash chain found in $HIST$, Alice can proceed immediately. This advantage becomes more apparent when we have a lot of data, and a user may need to lock multiple pieces of data at the same time to perform an operation.

In designing CloudLock, we wanted to avoid relying too much on the data owner to ensure correctness to create a new bottleneck. Our solution does not require the user to inform the data owner that the lock has been released in a strict fashion, only that the user informs the owner in some reasonable time period. For example, assume Bob was issued $WLOCK(3)$, but failed to inform the data owner when he released the lock. Later, Alice obtains $WLOCK(4)$ with the correct $HIST$. Then, when Alice releases $WLOCK(4)$, she informs the data owner. Since Alice will not accept that data unless the $HIST$ is correct, the data owner can interpret Alice's response as evidence the $WLOCK(3)$ has been released correctly.

B. Practical implementation

There has been recent interest [11], [12] in trying to provide additional services through a judicious application of an existing cloud provider's API. Here, we consider whether it is possible to implement a CloudLock-like solution on existing cloud services.

We based our sketch solution on Amazon's Simple Queuing Service (SQS) distributed messaging system [13]. SQS is a messaging system provided by Amazon to allow users to exchange messages between components using HTTP. A user can create *queues* that are identified by a URL to send and receive messages to and from other users. The design of SQS stresses availability and scalability (Amazon claims unlimited queue length and messages), but at the expense of consistency. For instance, querying SQS about the number of messages in the queue returns an approximate answer. SQS also does not provide FIFO guarantees. The key operations we use are provided below:

- *CreateQueue* Creates a queue that can be referred to by a unique URL.
- *SendMessage* Sends a message to the queue.
- *ReceiveMessage* Allows a user to retrieve some specified number of messages from the "top" of the queue.
- *ChangeMessageVisibility* Changes how long a message obtained from *ReceiveMessage* is blocked from other users.
- *DeleteMessage* Remove a specific message from a queue.

The *ReceiveMessage* has a property that once a user manages to read a message, that message is blocked from all other users for a time period determined by the *VisibilityTimeout* parameter. The user releases the message by calling *ChangeMessageVisibility* with $VisibilityTimeout = 0$.

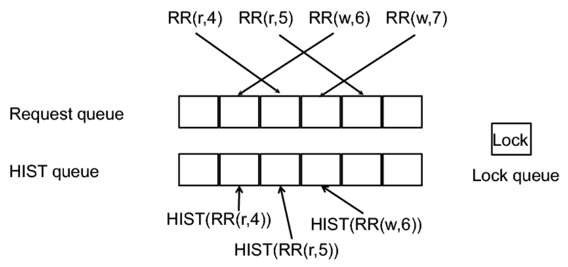

Figure 4. Sketch solution using SQS.

We let the data owner maintain three queues for each data: a request queue which stores all the pending requests; a history queue to store all the $HIST$ values; and a lock queue, which contains just one message. We have a rule that any user that wishes to do a write **must** be able to successfully call a *ReceiveMessage* from the lock queue. A user that wants to do a read *may* be able to do a read without successfully obtaining the lock queue message if the user is sure all other operations before it are reads. Fig 4 illustrates the setup. We see that the RR seqnum (which we will defined later) can be used to order the requests.

The intuition is to use the request queue to order the users request. The lock queue has only one message, and since a user that manages to read that message can block other users, the lock queue functions as a de facto conventional lock. We only have one message in the lock queue because SQS does not guarantee to return the latest message in the queue. By having only one message, we reduce the possibilities to either read the message (the user may have to poll multiple times to do this), or the message is blocked. The history queue is to verify that the data the user is currently reading is consistent with what the last user returned. For example, let there be two servers A and B in the cloud. At time 1, a user updates data to server A and when returning the lock, stores the $HIST$ in the history queue. Now, at time 2, another user grabs the lock and reads the data from server B. By repeatedly polling the history queue to obtain the appropriate $HIST$, the new user can verify that the copy of the data he obtains from server B is the same as the updated copy in server A.

We modify the central time server to return not a timestamp, but a request record (RR). A $(request, seqnum)$ pair where the request indicates it is a read or a write, and seqnum is an incrementing counter. So, Alice wanting to do a read, will be given RR $(r, 5)$. Bob, who is behind Alice and wants to do a write, will get RR $(w, 6)$.

Each user, after obtaining their RR, will check with the data owner to see if their RR is next in line. If it is, the user will do a *ReceiveMessage* from the lock queue to block other users, later calling *ChangeMessageVisibility* to continue blocking if necessary. When the user releases the lock, he will store a $HIST$ associated with his RR to the history queue.

Assuming the user does not grab the lock, the user will call *SendMessage* to the request queue and upload its RR. The user will then poll the request queue, trying to obtain a complete record of all the pending RR with a seqnum smaller than itself, but larger than the latest seqnum checked in with the data owner. So, assuming that the last seqnum the data owner records is $(*, 3)$ (the asterisk denotes either a read or a write request) and Alice has $(r, 5)$, Alice will poll the request queue until she obtains $(*, 4)$.

Now, if all the previous RRs before Alice were read locks, then Alice will go ahead and read the data, even if her *ReceiveMessage* to the lock queue is blocked. The reason is that Alice is now sure that all the other users before her are performing reads, so it is safe for her to do a read as well. She will also continue to try to call *ReceiveMessage* while she is still reading. However, if there is a RR with a write, then Alice has to wait until she can call *ReceiveMessage* successfully. Once Alice receives the message, she will query the history queue to obtain the latest $HIST$ uploaded. She can use the seqnum from her RR to check which is the appropriate $HIST$ value. Alice will then check whether the hash chain values match the DATA she obtained from the cloud. The reason why this additional step is necessary is that grabbing the message in the lock queue only ensures that no other user will access the data, but does not guarantee when that update is replicated throughout the cloud. After Alice releases the lock, she will update the history queue with her $HIST$ value.

To prevent Alice from competing for the lock message with a legitimate writer, we will require Alice to only call *ReceiveMessage* after the previous RR write has been checked in with the data owner. In our sketch solution, we require all users to be honest. Implementing CloudLock on an actual cloud provider is part of our on-going research.

While using three queues will incur an additional cost, we stress that not *all* data needs to be protected using locks. Less sensitive data can be accessed as before, and CloudLock is invoked only when sensitive data is read or written.

C. Other security threats

This paper focuses on defending an untrusted cloud provider. A different type of adversary we did not focus on is the *misbehaving user*. Similar to the misbehaving cloud, the misbehaving user is not malicious (otherwise, the owner will not authorize permissions), but may attempt attacks, such as deny committing an operation, or try to re-order the order of his operation through manipulating the lock numbers.

Since the user has to sign the $HIST$ with his private key when returning the assigned lock to the cloud, the cloud provider keep a copy of $HIST$ as evidence that the operation was performed by that user. Furthermore, the misbehaving user cannot replace either the read lock number or the write lock number, since this will result in an incorrect $HIST$ value.

Another attack the user can launch is a denial-of-service attack, where the user never relinquishes the lock. To defend against this attack, we can have the cloud provider contact the data owner if a user does not release the lock after a specified time period. The data owner can then invalidate that user's lock. For example, the data owner can create its own $UNLOCK - RLOCK(i)$ or $UNLOCK - WLOCK(k)$ that is signed with his own key, and instruct the cloud to forward this to subsequent users. For the remainder of the users, this is as if the data owner himself has been issued $RLOCK$ or $WLOCK$. Both the data owner and the cloud can take note to avoid committing any updates associated with the invalidated $RLOCK(i)$ or $WLOCK(k)$ if it ever appears later.

Another security threat we did not address is collusion. There are two types of collusion; *owner-user* collusion and *cloud-user* collusion. (The third type, *owner-cloud* collusion, appears to be unlikely.) In the owner-user collusion, the data owner and users collude to implicate that the cloud service provider violated some service level agreement (SLA) and demand compensation. CloudLock does provide some protection in that the use of signatures prevents the owner or users from creating fake lock numbers, but it is unclear what other types of attacks can arise from owner-user collusions. The problem of protecting cloud providers and defending against a cloud-user collusion remains an open problem.

D. Vector clocks

The use of the central time server in CloudLock can potentially create a bottleneck, since all users cannot access the cloud without first obtaining a time stamp. One possibility for removing the need for a central time server is to let the cloud and users each maintain a vector clock. A vector clock is a logical clock maintained by users. A causal ordering of user actions can be established through the exchange of vector clocks between users and the cloud.

The use of vector clocks raises two issues. The first is security. We want to prevent misbehaving users or the cloud from tampering with vector clock values to gain some advantage. This problem did not occur with the central time server since it is assumed to be trusted. One possible defense mechanism can be to let all changes be signed by the receiving party, and to let a history of prior vector clock with the appropriate signatures be maintained. The second issue is efficiency. There could be many users of a public cloud that do not communicate with each other frequently. Techniques to allow users to quickly update their vector clocks will be needed to improve performance.

V. Related Work

Much of the research on cloud computing security centers on untrustworthy cloud providers. On a general level, researchers have advocated techniques to audit the cloud [14], authenticate billing [15], and verify accountability to particular agreements [16]. Researchers have also studied cloud security for specific applications, such as healthcare [17]. Unlike these research, our work considers the security implications on an untrustworthy cloud performing a specific operation, maintaining data concurrency.

Other research that are more similar to our approach include work to verify the integrity of the hypervisor [18], check whether data has been deleted [19], ensure data indexing does not reveal sensitive information [20], or that the data has been correctly replicated [21]. Our focus on building a secure locking scheme can be viewed as a new, important tool for users to safely utilizing untrusted clouds.

Also relevant to our work is research by [11], [12], which seek to build reliable systems on top of an unreliable commercial cloud. The main difference is that their work assumes that the cloud provider is unreliable, but trustworthy. Our research considers an untrustworthy cloud that will manipulate its system for benefits. Work by [22] also considers an untrusted cloud, but focuses on proving the retrievability and not the consistency of the data.

VI. Conclusion

In this paper, we considered a cloud provider that seeks to provide consistency protection through the use of read and write locks. We first examined the potential attacks given the cloud's control of the locking algorithm, and then proposed a scheme to defend against these attacks. Our solutions are updates on classical locking algorithms to allow them to be executed by an untrusted cloud provider. In our future work, we intend to explore using vector clocks to replace the centralized time server, as well as to implement CloudLock on commercial cloud providers for evaluation.

Acknowledgments

This research was supported in part by NSF grants CCF 1028167, CNS 0948184, and CCF 0830289.

References

[1] P. Noordhuis, M. Heijkoop, and A. Lazovik, "Mining Twitter in the Cloud: A Case Study," in *IEEE CLOUD*, 2010.

[2] P. Teregowda, B. Urgaonkar, and C. Giles, "Cloud computing: A digital libraries perspective," in *IEEE CLOUD*, 2010.

[3] http://www.cio.gov/, "State of Public Sector Cloud Computing."

[4] M. Armbrust and et al, "Above the Clouds: A Berkeley View of Cloud Computing," EECS Department, UC-Berkeley, Tech. Rep., 2009.

[5] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina, "Controlling Data in the Cloud: Outsourcing Computation without Outsourcing Control," in *ACM CCSW*, 2009.

[6] M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono, "On Technical Security Issues in Cloud Computing," in *IEEE CLOUD*, 2009.

[7] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Financial Cryptograpy (FC)*, 2010.

[8] E. Shi, J. Bethencourt, T.-H. H. Chan, D. Song, and A. Perrig, "Multi-dimensional range query over encrypted data," in *Oakland*, 2007.

[9] J. Wu, *Distributed System Design*. Boca Raton, FL, USA: CRC Press, Inc., 1998.

[10] C. Gentry and A. Silverberg, "Hierarchical ID-Based Cryptography," in *ASIACRYPT*, 2002.

[11] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, "Building a Database on S3," in *ACM SIGMOD*, 2008.

[12] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer, "Making a Cloud Provenance-Aware," in *First Workshop on on Theory and Practice of Provenance*, 2009.

[13] http://aws.amazon.com/sqs/, "Amazon Simple Queue Service."

[14] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson, "Security Audits of Multi-tier Virtual Infrastructures in Public Infrastructure Clouds," in *ACM CCSW*, 2010.

[15] K.-W. Park, S. K. Park, J. Han, and K. H. Park, "THEMIS: Towards Mutually Verifiable Billing Transactions in the Cloud Computing Environment," in *IEEE CLOUD*, 2010.

[16] A. Haeberlen, "A Case for the Accountable Cloud," in *ACM SIGOPS LADIS*, 2009.

[17] R. Zhang and L. Liu, "Security models and requirements for healthcare application clouds," in *IEEE CLOUD*, 2010.

[18] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity," in *ACM CCS*, 2010.

[19] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman, "FADE: Secure Overlay Cloud Storage with File Assured Deletion," in *Securecomm*, 2010.

[20] A. Squicciarini, S. Sundareswaran, and D. Lin, "Preventing information leakage from indexing in the cloud," in *IEEE CLOUD*, 2010.

[21] R. Curtmola, O. Khan, R. Burns, and G. Ateniese, "MR-PDP: Multiple-Replica Provable Data Possession," in *IEEE ICDCS*, 2008.

[22] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A High-availability and Integrity Layer for Cloud Storage," in *ACM CCS*, 2009.