2015 IEEE Symposium on Security and Privacy

Caelus: Verifying the Consistency of Cloud Services with Battery-Powered Devices

Beom Heyn Kim* and David Lie†
*Department of Computer Science
†Department of Electrical and Computer Engineering
University of Toronto

***Abstract*—Cloud storage services such as Amazon S3, DropBox, Google Drive and Microsoft OneDrive have become increasingly popular. However, users may be reluctant to completely trust a cloud service. Current proposals in the literature to protect the confidentiality, integrity and consistency of data stored in the cloud all have shortcomings when used on battery-powered devices – they either require devices to be on longer so they can communicate directly with each other, rely on a trusted service to relay messages, or cannot provide timely detection of attacks.**

We propose Caelus, which addresses these shortcoming. The key insight that enables Caelus to do this is having the cloud service declare the timing and order of operations on the cloud service. This relieves Caelus devices from having to record and send the timing and order of operations to each other – instead, they need to only ensure that the timing and order of operations both conforms to the cloud's promised consistency model and that it is perceived identically on all devices. In addition, we show that Caelus is general enough to support popular consistency models such as strong, eventual and causal consistency. Our experiments show that Caelus can detect consistency violations on Amazon's S3 service when the desired consistency requirements set by the user are stricter than what S3 provides. Caelus achieves this with a roughly 12.6% increase in CPU utilization on clients, 1.3% of network bandwidth overhead and negligible impact on the battery life of devices.

I. Introduction

A commonly available type of cloud computing service is a cloud storage service, which offers persistent and highly-available storage over the Internet. Such services include basic object storage services such as Microsoft Azure Storage or Amazon S3, personal storage services as DropBox, Google Drive or Microsoft OneDrive, and database services such as Amazon RDS and DynamoDB and Microsoft Azure SQL Database. These services are popular because they offer users many useful attributes, such as backup and versioning for data, automatic scaling and failure recovery, replication and access to data across devices and the ability to collaboratively share data with other users. Industry figures indicate that there has been rapid increase in the use of such cloud services. For instance, Google Drive currently boasts over 10 million users [1].

However, cloud services can be a threat to the security of their users. While we believe it is unlikely that a cloud service provider would deliberately attack its customers, using a cloud service still exposes users to new threat. Cloud services provide service to multiple, often mutually distrustful, users on a shared infrastructure. Vulnerabilities in the infrastructure may allow a malicious user to compromise parts of the cloud service and attack another user. In addition, cloud services' employees, as part of their duties, may have privileged access to parts of the infrastructure or the software that implements the infrastructure. While good industry practice often means that no particular employee will have access to the entire infrastructure, there has been evidence that strategically placed insiders have been used by certain organizations to attack cloud users [2]. Thus, while a cloud service on as a whole might not be malicious, the component that stores and serves user data can be compromised by both external and internal attackers, which can threaten the confidentiality and integrity of the data stored on the cloud service by the user. For brevity, we will henceforth refer to a cloud service whose data storage component has been compromised as simply a "malicious cloud service" even though there could be components of the service that are not compromised.

Since cloud storage services are implemented as globally distributed systems designed to provide different devices or users access (even concurrent access) to the same data, there is an implicit or explicit assurance of consistent access according to some *consistency model*. A consistency model defines acceptable delays between when the results of an operation by one device become visible to other devices, as well as the order in which those operations should become visible. Applications using the cloud storage service are implemented based on the consistency model. Therefore, applications will misbehave when the assumption on the consistency model is violated. By omitting, reordering, replaying and truncating operations, the malicious attacker can mount subtle *consistency* attacks to cause violation of delay and ordering constraints of the consistency model. Such attacks can seriously damage applications composed of distributed processes collaboratively interacting with each other across devices via the cloud storage service, because the application's behavior depends on the ordering and timing of the previous operations. For example, source code repository such as Git repository may suffer from truncated operations (*fork* attack [3]–[5]) which cannot be resolved by the built-in hash chain mechanism. In addition, an authorization service may inadvertently reveal sensitive information to unauthorized users when a revocation request is maliciously delayed or dropped after the read and update operations. Such an attack would allow revoked users to access data they should not have permission to access.

DOI 10.1109/SP.2015.59

While there have been several recent proposals for protecting users from such attacks, they suffer from deficiencies in terms of security, battery-friendliness or timely detection. For example, a number of approaches use an external service such as email or instant messaging to enable devices to exchange messages about data they have stored on the cloud service [5], [6]. Unfortunately, while this defends against a malicious cloud service, it relies on the external service being trusted to reliably deliver messages. Other approaches eschew an external service in favor of having clients communicate directly with each other in a peer-to-peer fashion [4], [7] or they rely on a highly-available device to broadcast information to all devices [3], [8]. However, such an approach is not battery-friendly as such devices must be awake all the time to communicate and a peer-to-peer approach causes devices to consume more energy due to increased network traffic. An increasingly large percentage of user devices are battery-powered, so negative impact on battery life is a serious drawback. Finally, some approaches such as CloudProof [9] advocate infrequent intervals where logs of operations will be collected from all devices at an auditing service. Naturally, such infrequent intervals preclude timely detection.

Furthermore, all existing proposals can each only check a single consistency model, but cloud services offer a variety of consistency models. For example, some proposals only check strong consistency [3], [7]–[9] or while others only check causal consistency [4]–[6]. None have been demonstrated to be general enough to be used to check several consistency models.

In this paper, we present Caelus, which overcomes these shortcomings. The key insight that enables Caelus to do this is having the cloud service declare the timing and order of operations on the cloud service. This relieves Caelus devices from having to record and send the timing and order of operations to each other – instead, they need to only ensure that the timing and order of operations both conform to the cloud's promised consistency model and that it is perceived identically by all devices. To do this in a secure, battery-friendly and timely manner, Caelus employs several novel mechanisms.

First, Caelus detects inconsistencies in near real-time by having an *attestor* sign (i.e. attest) the order and timing of operations declared by the cloud service. These *scheduled attestations* are written back to the cloud storage service every few seconds according to a pre-determined schedule that is known to all devices. Other devices can then read these attestations from the cloud service and use them to verify the consistency of the operations they have performed without having to directly communicate with each other or the attestor, thus reducing network usage and battery drain. In addition, because an attestation for an operation is available within seconds, devices can perform timely detection of consistency violations. Moreover, Caelus' protocol guarantees a malicious cloud service cannot subvert Caelus by dropping or delaying attestations and as a result, the storage and distribution of attestations need not be performed by a trusted service.

Second, in its most basic instantiation, the attestor is a single device that is actively signing attestations every few seconds, but this would not be battery-friendly for the attestor. To reduce the impact on battery life, Caelus introduces *attestor-partitioning* which partitions the attestor into a single *root attestor* device that must be periodically accessible but can otherwise sleep to conserve battery and an *active attestor* that must be active but whose role should be assigned to a device already active for other reasons (i.e. it is writing data or the user is already using it). Caelus ensures that even though the attestor role may be distributed across several devices who only communicate via the cloud service, a malicious cloud service cannot partition the devices into groups that are out of sync with each other.

Finally, to enable the detection of violations for different consistency models, Caelus modularizes the task of verifying that responses from the cloud service into a *consistency verification procedure* that can be performed independently by each device. This allows the same Caelus system to be used to check different consistency models and therefore Caelus can verify cloud storage systems that provide strong, causal and eventual consistency models. In addition, the distribution of these checks across devices means that no single device will become a bottleneck as the number of devices increases.

We make the following contributions in this paper:

- We present the design of Caelus, which uses *scheduled attestations* for verifying the consistency of a cloud service. We describe our Caelus prototype that runs on Amazon's S3 storage service, and demonstrate that we can detect consistency violations when we ask Caelus to bound inconsistency to a period shorter than what S3 can provide.
- We show that *attestor-partitioning* can reduce the battery drain of Caelus on the attestor by about 40× without reducing the security of the system. Our measurements show that Caelus increases CPU utilization on clients by about 12.6% and imposes network bandwidth overhead of about 1.3%. Under normal, failure-free scenarios with an honest server, the user should experience no perceptible overhead or loss of availability as a result of Caelus.
- We provide three *consistency verification procedures* that enable individual devices to use Caelus to verify strong, eventual and causal consistency using a series of logical checks over a signed log of operations.

We start with a couple of potential but realistic consistency problems in Section II. Then, we discuss Caelus' security model and assumptions as well as the guarantees it provides in Section III. The various design aspects of Caelus are discussed in Section IV. Then, we present an analysis of Caelus' security properties in Section V. Section VI describes implementation details of our prototype and the following Section VII gives evaluation results of our Caelus prototype. We then discuss related work in the Section VIII and finally conclude in the Section IX.

II. Motivating Scenarios

To motivate the seriousness of consistency attacks, we describe two common scenarios where a malicious cloud service can subvert the victim software systems.

Git Repository. In the first example, consider a user or group of users that use an online Git repository hosted in the cloud such as Github[1]. Git repositories should be strongly consistent – individual commits need not be totally ordered but set of commits pushed to the Git service should be visible to every client as soon as the push is completed thus forming a total order of pushes and enabling consistent conflict resolution. Git uses hashes to verify the integrity of the commit history. The hash of each commit depends on the parent commit thus forming a hash chain. Also, each Git client keeps a complete history of all commits by all others. Thus, it might initially appear that Git's hash chain generated by the Git service and local copies of the commit history stored by each client prevent the misbehavior by the Git service.

However, by subverting the timing requirement that all operations should be made visible to all other requirements as soon as the operation completes so that some commits are *never* made visible, a malicious Git server can corrupt the source code repository in a way that is not detectable by individual clients. As a result, a malicious service can hide a particular push and all future pushes by one or more clients from a different set of one or more clients. This effectively partitions or forks the group of clients into two or more sets that are unaware of each other's pushes, but are also unable to tell whether the other group has not committed anything or whether the server is maliciously dropping commits without directly communicating or using a trusted service to communicate.

Authentication Service. A security sensitive operation in federated identity and authorization services such as OAuth is credential revocation. OAuth is used in a variety of online services such as Google and Facebook to authorize untrusted parties (relying parties in OAuth parlance) to access information belonging to the user. In OAuth 2.0, revocation requests should be processed and propagated to all servers immediately so that the revocation takes effect as soon as possible [10]. If an OAuth implementation uses a cloud storage service, it will depend on the cloud service providing consistent update of revocation requests to all application servers. For example, consider a user that revokes access to a document to an untrusted party before adding sensitive information to it. Since the revocation happens before the addition of sensitive information, a cloud service that promises strong or causal consistency should ensure that all servers see the revocation request before the addition of the sensitive information, i.e., in the same order. However, a malicious cloud service may break this promise and replicate the operations in an inconsistent order. As a result, some nodes may receive the revocation request after the addition of the sensitive information and reveal the sensitive information to the revoked parties, even though this contradicts the user's expectations.

[1] https://github.com/

III. Security Model and Guarantees

Caelus is designed for any user who owns multiple Internet devices, some of which are battery-powered devices, such as tablets, smartphones or laptops, and which may have wireless network connections that can fail. While Caelus can support non-battery powered devices that have reliable network connections, Caelus uses special mechanisms to mitigate power consumption and network failures. We envision that many devices of the future, such as smart-home devices, smart-cars and Internet-of-things devices will also have wireless network connections and be battery-powered. We begin by stating our security model which describes our assumptions regarding the behavior and capabilities of clients, the network and the cloud service. We then state the security guarantees that Caelus provides.

A. Security model

Clients. Clients are devices that are under the user's control and are used to access data stored on the cloud service. For example, a client can be the individual user's laptop, tablet, or other battery-powered device. A client can also be a device owned by other users with whom the primary user has shared access to the cloud service. We assume clients can become malicious or unavailable for a variety of reasons. They can become malicious due to infection by malware, compromise or theft. Likewise, they can become unavailable due to software failure, loss of network connectivity, loss of battery power or system sleep to conserve power. In cases where impending unavailability is known beforehand, such as a system sleep or battery depletion, the client can warn the other devices allowing them to take actions to mitigate the effect of the unavailability.

We assume that each client has a public-private key pair that can be used for digital signatures, and that the public keys of clients are known to other clients and to the cloud provider. To protect against man-in-the-middle attacks, we assume public keys are either distributed using a protected channel or a PKI exists to certify their authenticity. Each user also has an encryption key that is used to encrypt the user's data to protect it from disclosure to the cloud service. We also assume a secure key distribution mechanism for the shared encryption key so that it is only shared with the user's clients and the clients that the user is sharing data with.

We assume that clients have reasonably synchronized clocks. The degree of clock synchronization required depends on the accuracy at which the user wants to detect a malicious cloud server. While previous work has shown that very highly synchronized clocks are possible [11], for storage with personal data, we believe that limiting clock skews to several milliseconds, such as that which can be achieved using NTP, should be sufficient.

Network. We assume a network model that provides connectivity between each client and the cloud service, but does not

provide direct connectivity between clients. In addition, the network may fail to transmit messages between client and the cloud service and clients cannot distinguish between a failure of the network and a failure of the cloud service. Assuming that all communication between clients must traverse the cloud service enables clients to communicate even if both are not online simultaneously [12].

Cloud Service. The cloud service promises a certain consistency model for data stored on the cloud service. An honest cloud service will respond to requests for data from various clients according to the promised model. Caelus further assumes that cloud services offer a time bound on consistency models, which means that operations are guaranteed to become visible to all clients within some *visibility time bound*, which is specified by the cloud service provider in SLA. In practice, consistency models that are not bounded are less useful because it is very hard to reason about the data when developing client software. Furthermore, unbounded consistency models can result in unresolvable conflicts. As a result, recent work has shown that in practice, most systems that claim to be weakly consistent are still bounded [13]. In fact, there are a number of proposals in the literature that enable users to measure the time bound that a cloud service offers [14]–[20]. Thus, bounded consistency models are realistic and we believe cloud service providers may even be motivated to claim shorter time bounds than their competitors.

In our security model, a malicious cloud service's goal is to violate the promised consistency model and trick the user into unknowingly using inconsistent data, or alternatively, to claim a consistency model stronger than what they offer and hope that the user won't notice the discrepancy. A malicious cloud service can selectively omit, replay, reorder or delay the results of operations by clients. In addition, since all client communication goes through the cloud service, the cloud service can also selectively fake client failures by preventing operations made by a client from becoming visible to one or more other clients. However, we assume that standard cryptographic assumptions hold – a malicious cloud provider cannot decrypt data for which it doesn't know the key, nor can it forge cryptographic signatures. In other words, we use the Dolev-Yao attack model for a malicious cloud service.

Similar to clients, we assume that the cloud service has a public-private key pair and that the public key is well-known to all clients. Thus, the cloud service's response signed with cloud's private key is non-repudiable.

Collusion. As we will discuss in Section III-B, Caelus makes security guarantees against both a malicious cloud service and malicious clients. Caelus assumes that malicious clients can collude and defends against them. However, if the clients and cloud provider are both malicious and collude, it would be difficult to make any guarantees since there are no non-malicious components left in the system. As a result, we weaken the security model slightly by assuming that clients are *cloud-secure*, meaning that they can be compromised and act maliciously, but are always secure against compromise by the cloud service. For example, the cloud-secure assumption holds if clients are infected with malware or have been stolen, so long as that malware or the thief is not under the control of the cloud service.

We believe this assumption is realistic for several reasons. First, many cloud services provide APIs for developers to develop their own client software [21]. For example, there exist a plethora of 3rd party DropBox clients that enable users to automatically backup their files, synchronize data or use multiple backup services [22]. As a result, the provenance of the client software is largely independent of the cloud service provider.

Second, in cases where the user is using a client provided by the cloud provider, there can still be independence if the client software and cloud storage service are hosted on separate systems. Thus, an attacker who compromises the cloud storage service does not automatically get the ability to corrupt or control the client software.

B. Security guarantees

We now state the security guarantees that Caelus provides. Because our security model allows for both a malicious cloud service and malicious clients, we separately describe the guarantees that hold against each of these.

Caelus provides the following security guarantees against a malicious cloud service:

SRV1: A malicious cloud provider cannot read user data.

SRV2: A malicious cloud provider cannot tamper with user data without being detected.

SRV3: A malicious cloud provider that responds inconsistent data will be detected within a finite time bound defined by T_{Caelus}.

Against malicious clients, Caelus provides a different set of guarantees. Since devices have the ability to read and modify data, Caelus cannot protect the confidentiality or integrity of data on the cloud against a malicious device. This could be somewhat mitigated by access control, but the amount of protection would still be dependent on the access control policy so it cannot provide complete protection for data confidentiality and integrity. We thus leave the integration of access controls into Caelus for future work.

However, since all operations must be signed, Caelus does guarantee that operations by clients are non-repudiable. In addition, a malicious client may attempt to falsely accuse the cloud provider of violating consistency guarantees. Caelus guarantees that such false accusations can be invalidated using an audit procedure. In summary, Caelus provides the following guarantees against malicious clients (including multiple colluding clients):

CLT1: Malicious clients cannot repudiate modifications they have made to data on the cloud.

CLT2: Malicious clients cannot falsely accuse the cloud service of violating the promised consistency model.

Caelus does not protect against loss of data against a malicious cloud provider. A malicious cloud provider can

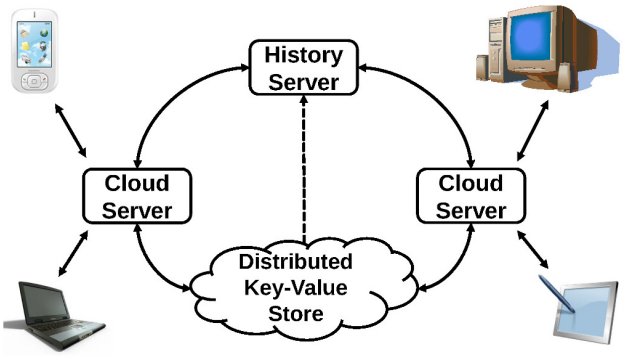

Fig. 1. Caelus architecture.

always drop a user's request or destroy data after receiving it. Similarly, a malicious client can overwrite data or refuse to perform its duties, thus affecting the durability and availability of user data. However, in the absence of benign failures or malicious activity, Caelus provides the following guarantee:

AV1: Under normal operation where clients and the cloud service are free of failures and malicious activity, Caelus will not cause delays or unavailability of the cloud service.

Since Guarantee AV1 doesn't hold if there are any malicious parties, it should be clear that it is really more of a performance guarantee rather than a security guarantee. However, we believe this guarantee is still important since to be practical, Caelus' security guarantees should impose little or no cost under normal (non-malicious) circumstances.

IV. Design

A. System overview

Caelus is a set of enhancements that can be added to a cloud service that uses a distributed key-value store. We select this storage architecture because essentially all cloud storage services are based on key-value stores at their lowest level [23]. The architecture of a typical Caelus system is illustrated in Figure 1. The existing cloud service contains a set of geographically distributed *cloud servers*. Clients connect to a cloud server that is close to them for low latency. Aside from clients, cloud servers communicate with two components: an existing globally distributed key-value store that provides some consistency guarantees and a new centralized *history server* that is added by Caelus. Enhancing an existing cloud service to support Caelus generally entails adding the history server and modifications to the cloud servers and clients, but does not require changing the distributed key-value store. Moreover, deploying the Caelus client to customer devices can be accomplished by having users install client software equipped with the Caelus verification scheme. The security of their data is one of the main concerns cloud users have, and as a result, we believe cloud service providers may be motivated to deploy Caelus to convince users that their data is safe, and to remove legal liability from themselves as Caelus guarantees hold even if the cloud infrastructure is compromised

Although it is under the control of the cloud service and is not trusted by the user, the history server plays a crucial role in Caelus. Instead of having the devices assemble a view of all operations that have taken place and check the consistency of each operation, the history server has the cloud service declare the history of operations it has performed. Then, all the devices have to do is verify that 1) the declared history conforms to the promised consistency model and that 2) all the operations they have performed are reflected in the log. This considerably simplifies the consistency verification procedure, enables it to be distributed across devices and eliminates the need for devices to communicate directly with each other. The main guarantee that Caelus must provide then is that all devices perceive the same declared history from the cloud service.

Clients read and write data from the cloud service with `Get` and `Put` operations. Each cloud server forwards the operations from clients connected to it to the key-value store. The history server records a log of all operations that have occurred on the cloud service. Cloud servers forward `Gets` to the history server as soon as they are received from clients. However, the key-value store is globally distributed so there is a delay between the time that a cloud server accepts a `Put` from a client and forwards it to the key-value store and the time that the result of the `Put` has been made visible to all other cloud servers. Thus, `Puts` are only logged by the history server when it has been notified by the key-value store that they have been made globally visible. If the key-value store is not capable of such notifications, the history server can also log the `Puts` on behalf of the key-value store after the visibility time bound has passed. The order that operations are logged on the history server is unimportant – instead, clients rely on embedded timestamps in operations to reconstruct order. The history server is also responsible for storing control messages that are sent between clients, such as attestations and selection.

We begin by describing a *basic system* that uses a single monolithic attestor and provides all the security guarantees described in Section III-B. However, the basic protocol is not battery-friendly so we then describe an enhanced *battery-friendly system* that uses attestor-partitioning to enable devices that are not being actively used to sleep and conserve energy. Finally, we will discuss some operating parameters of Caelus.

B. Basic system

We describe our basic system in four steps. First, we describe how `Put` and `Get` operations are implemented by the cloud server. Second, we describe the attestation procedure Caelus uses to ensure that every client has an identical view of the history of operations. Third, we describe how each client verifies that its local view of operations is consistent with the attested history of operations. Finally, we describe how clients join and leave the Caelus system. The major elements of the Caelus protocol are illustrated in Figure 2.

Operations. Each `Get` and `Put` operation transmits the following meta-data in the header: operation type (`Get` or `Put`), key value, client ID, a timestamp, a sequence number and a hash of the data if it is a `Put` operation. The entire

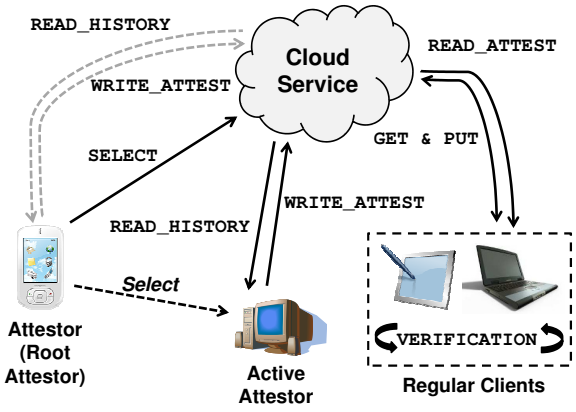

Fig. 2. The Caelus protocol. The Root Attestor, selects the Active Attestor using the `Select` operation. The Active Attestor then reads the history using `Read_History`, and signs the history to produce attestations, which are written back using `Write_Attest`. When no Active Attestor is available, the Root Attestor can perform attestation itself (shown in gray). Regular clients perform `Get` and `Put` operations and verify these operations by reading attestations using `Read_Attest` and running the verification procedure on the attested histories.

header is signed with a private key specific to each client, and whose matching public key is known to all other clients. The sequence number and hash are used to detect omissions, replay and tampering of data. The timestamp is used by clients to reconstruct the order and timing of events. Any data transmitted in a `Put` is encrypted by the device to enforce Guarantee SRV1 and both the header and any data are signed by the device to enforce Guarantee SRV2 and Guarantee CLT1.

The cloud servers do not buffer any data; their main purpose is to provide a single interface to the clients and hide the details of the key-value store and history server from the clients. Client `Put` and `Get` requests are directly forwarded to the globally distributed key-value store and the results of `Gets` are returned back to clients by the cloud servers. `Gets` forwarded to the history server are logged immediately while `Puts` are only logged after the key-value store notifies it that the results of the `Put` have become globally visible or the visibility time bound has passed. The history server assigns global sequence numbers to logged operations, which are only used as a way for clients to request sections of the log. `Puts` that have been received but are not yet logged are not assigned global sequence numbers and are not yet visible to every client. While the history server is shown as a single machine in the Figure 1 it need not be. However, if distributed, one important caveat is that each operation must be assigned a unique global sequence number, so the history server must be at least sequentially consistent. This requirement only holds for keys that have common clients. If two sets of keys do not have any clients in common, the assignment of sequence numbers among those sets need not be sequentially consistent. Violating this requirement will result in clients detecting consistency violations, as it will either result in operations with duplicate sequence numbers. Thus, it is of no benefit for a malicious cloud service to violate this requirement. In addition, the history server only stores hashes of data objects instead of the full data objects, so the amount of data stored is relatively small.

The log on the history server is intended to act as "proof" that the cloud service is adhering to its promised consistency model. Depending on the consistency model, the servers and key-value store may also include additional information about each operation to facilitate the verification that the consistency model is met. We discuss the details of the consistency verification procedures below.

Attestation. One of user's devices, acting as the attestor, periodically performs attestations by fetching a log segment from the history server. For now, we assume the attestor has no battery limitations, which we will remove by employing *attestor-partitioning* protocol described in section IV-C. There are two requirements that the attestor must meet. First, the role of the attestor is permanently assigned to one and only one device and the identity of the attestor should be known to all other devices. Second, the attestor should periodically perform attestation operations on a schedule that is also known to all devices.

To request a log segment, the attestor uses a `Read_History`(G_{Start}, G_{End}) operation, which specifies a section of the log between two global sequence numbers G_{Start} and G_{End} to read. The attestor submits this request to the server it is connected to, which reads it from the history server and returns the results to the attestor. All log segments are signed by the history server to ensure that a malicious client cannot tamper with them, which enforces Guarantee CLT2. To create the attestation, the attestor adds a sequence number and timestamp to the log segment, signs it and stores it back to the history server a `Write_Attest`(G_{Start}, G_{End}) operation. Clients can read attestations from the history server by using a `Read_Attest`(G_{Start}, G_{End}) operation, which returns all operations and attestations in the requested range. The attestor performs attestations at specific time intervals defined by the parameter T_A.

Clients expect to be able to read a new attestation every $T_A + \epsilon$. ϵ accounts for variable delays due to network and processing and must be added any time a client is measuring the delay between two events on the cloud service. This *scheduled attestation* prevents a malicious service from showing different log contents to different clients. If a malicious service tries to tamper with or drop portions of the log, it will be detected when clients verify the log segments against the attestations. Replay or omission of log segments or attestations will be detected by missing sequence numbers in the stream of attestations. Finally, a malicious service may attempt to drop all future log segments and attestations (i.e., truncation), but this will be detected because clients will not be able to read an attestation at the expected time. Clients cannot distinguish

between this type of malicious cloud service and a failed attestor, but since the attestor is available most of the time and assumed to only experience failures for short periods of time, Caelus clients halt until they are able to read any missed attestations. If a client continues to miss attestations for an extended period of time, it can notify the user who can then examine the state of their attestor device to determine if the device or the device's network connection has failed or not. If neither the device nor its network has actually failed, this indicates that the cloud service is acting maliciously.

Using scheduled attestations, all clients can safely assume that all attestations will be eventually made identically visible to all clients. By extension, this guarantees that all clients will see the same history of operations and from this, detect if the cloud service is maliciously trying to omit, reorder or delay client operations using the verification procedure we describe next.

Verification. To distribute the verification tasks, each client is responsible for verifying the consistency of its own operations. Verification happens asynchronously to `Put` and `Get` operations when clients periodically fetch attestations using the `Read_Attest` operation. Caelus verifies that operations are inconsistent by at most some time bound T_{Caelus}, thus enforcing Guarantee SRV3.

Clients verify their operations in 3 steps. First, clients verify the correctness of the fetched log segment against the accompanying attestation. Second, clients perform a *presence check*, where they verify the individual signatures on each operation in the log to detect tampering, and check that the log segment does not omit or replay operations using the sequence numbers embedded in the operations. Finally, clients verify the consistency of their `Put` and `Get` operations. The exact method that clients use to verify the consistency of `Puts` and `Gets` depends on the consistency model of the cloud service.

Caelus currently supports 3 consistency models: strong consistency, eventual consistency and causal consistency with some time bound defined by the *visibility time bound* T_S. Under strong consistency, all operations appear to execute in a single global order with every `Get` receiving the value of the immediately preceding `Put` to the same key. In addition, all `Puts` should be globally visible as soon as they are acknowledged by the cloud service. This makes the verification of strong consistency the simplest of all three models. Clients verify the consistency of `Puts` by checking that the `Put` appears in the next attestation signed by the attestor. This means that a cloud service could at most delay the effects of a `Put` by $T_A + \epsilon$. Clients verify the consistency of `Gets` by checking that the value returned matches the value of the immediately preceding `Put` in the log.

In the bounded eventual consistency model, the results of `Puts` need not be immediately visible to all clients, but may instead take up to the visibility time bound T_S, to become visible to all clients. This is equivalent to the definition of bounded consistency used by Pileus [24]. The checks that clients do to verify consistency are illustrated in Figure 3. To

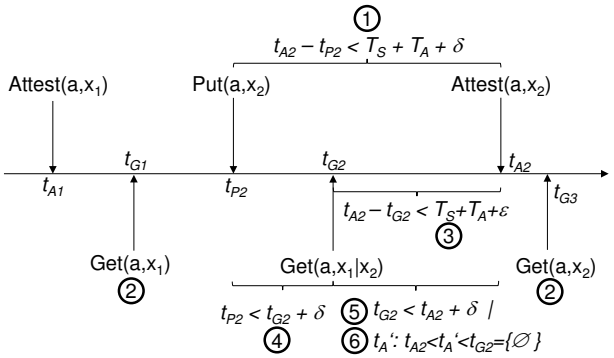

Fig. 3. Verification of eventual consistency. We denote operations as $operation(key, value)$. $x_1|x_2$ means that the `Get` may legally return either x_1 or x_2.

verify the consistency of `Puts`, the client will check that ① the attestation time of all of its `Puts` are at most $T_S + T_A + \delta$ after the corresponding `Put` has been acknowledged by the cloud service, where δ accounts for clock skew between clients when comparing timestamps.

Checking the consistency of `Gets` is slightly more complex. At time t_{A1}, the attestor attests a `Put` with value x_1. Then at time t_{P2}, a client performs a `Put` with value x_2 to the same key, but because of the consistency model, x_2 is not globally visible and attested until time t_{A2}. The `Get` at t_{G1} must return x_1 because x_1 has been attested and is thus globally visible, while the `Get` at t_{G3} must return x_2 for the same reason. However, the `Get` at t_{G2} may return either x_1 or x_2. Thus, to verify the consistency of a `Get`, ② the client first checks whether the value returned by the `Get` matches the most recent attested value. If not, it is either a violation or the `Get` has read the value of a `Put` that has yet to be attested. The client maintains a list of unverified `Gets` and ③ waits for an attestation for a matching `Put` to appear within the *timeout period* $(t_{A2} - t_{G2}) < T_S + T_A + \epsilon$. Note that within this time, the client can use the value of the `Get` as any violation will be detected within the timeout period. If the attestation does not appear before the timeout period, either the cloud service is taking too long to replicate results or the service returned stale results, both of which are consistency violations. If an attestation for a `Put` does appear in time, then the client checks that ④ the timestamp of the `Get` is later than the timestamp of the `Put`, i.e. $t_{G2} > t_{P2} - \delta$. It must also check that either the `Get` ⑤ is before the `Put`'s attestation, i.e. $t_{G2} < t_{A2} + \delta$ or ⑥ if the `Get` is after, that there are no newer attested `Puts` that the `Get` should have read, i.e. $t'_A : t_{A2} < t'_A < t_{G2} = \{\emptyset\}$. Check ⑥ handles the case where the attestation happens before the `Get`, but is not fetched and verified by the client until after the `Get`. If the `Get` passes these checks, then it is verified and removed from the unverified `Get` list. Otherwise, the `Get` remains on the list and will be checked against other attested `Puts` until either it is verified or it times out and becomes a violation.

A cloud service that implements bounded causal consistency for Caelus enforces causal consistency on the values read

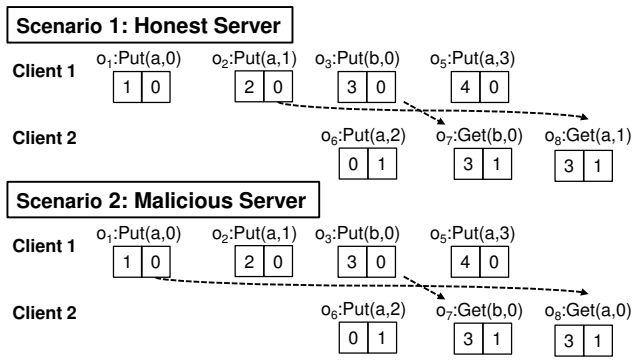

Fig. 4. Verification of causal consistency. The vector clock is shown below each operation as [C1 | C2].

by Gets, and will eventually make all Puts visible to all clients via the history log. Bounded causal consistency is also referred to as Causal+ consistency [25] in the literature. Because Puts must be made globally visible in a bounded amount of time, verifying the consistency of Puts in bounded causal consistency is the same as verifying Puts in bounded eventual consistency.

As with eventual consistency, some Gets may see the result of Put operations before they become globally visible in the log. Thus, clients perform the same verification steps in causal consistency as eventual consistency. However, while a Get in eventual consistency may return either the most recently attested value or *any* written but unattested value, Gets in causal consistency must return the most recent value on which it is causally dependent. One option for verifying Gets would be for clients to extract the chain of causal operations it is dependent on and then verify that the value read matches that of the most recent Put in the chain. However, if the client only knows the value that the Get read, it may be ambiguous which Put it is actually dependent on if there are several Puts with the same value.

To uniquely identify each operation, we enhance the cloud servers to attach a vector clock to each operation in the log [26]. Clients verify the correctness of the vector clocks by checking that they increase along with the sequence numbers on operations, which indicate program order. Clients can then use the vector clocks to verify the freshness of the value read by checking if there are any newer Puts to the same key between the vector clock of the Get and its associated Put. Like in eventual consistency, a client may have to defer verification for up to $T_S + T_A + \epsilon$ to ensure all necessary attestations have occurred.

To illustrate, consider Figure 4. The Get, o_8, by client $C2$ reads the result of the Put, o_2, by $C1$. We denote a vector clock of an operation using the notation $vc(o_i)$. Note that vector clocks only increase on Put operations. $C2$ verifies the consistency of the Get by verifying that there are no Put operations on the same key with vector clocks greater than o_2 and less than o_8. In Scenario 1, there is no violation because all operations between o_2 and o_8 modify other keys. o_1, o_5 and o_6 modify the same key, a, but since $vc(o_1) < vc(o_2)$, $vc(o_5) \parallel vc(o_8)$ and $vc(o_6) \parallel vc(o_2)$ ($\parallel$ means "incomparable", the two values have no defined order), their results may legitimately be invisible to o_8. However, in Scenario 2, the cloud service, either maliciously or erroneously, returns the result of o_1 instead of o_2 to o_8. In this case, client verification will fail because it finds that $vc(o_1) < vc(o_2) < vc(o_8)$ and that o_2 is a Put to the same key read by o_8. A malicious cloud service cannot assign o_2 a vector clock less than $vc(o_1)$ because the order of the vector clocks must match the sequence number embedded in the operations. Neither can it omit o_2 since the presence check done by clients will detect that the o_2 operation is missing from the log.

For large numbers of clients, vector clocks can be expensive since the length of the vector is determined by the number of nodes in the system [25]. However, in Caelus vector clocks do not need to span users who do not share data. Instead, the size of the vector only has to accommodate the number of clients a user has (or is sharing data with), which we expect to be generally fewer than 10-20.

Client join and leave. When a client joins or rejoins the system after a period of being asleep, it must verify that the attestor is available before performing any operations. It does this by checking that the timestamp of the most recent attestation posted by the attestor is less than $T_A+\delta$ old. Once it establishes that the attestor is available and making attestations properly, it can proceed to access values on the cloud service.

If the client has been disconnected from the cloud service for a long period of time, it may have to download a significant portion of the log to verify the consistency of Gets that read values written many operations ago. To bound the length of the log, the attestor can periodically checkpoint the entire key-value store by performing a Get and Put on every key, attesting to the new key values and having the history server discard all log entries before the checkpoint. To safely checkpoint a key, Caelus must ensure that there are no Puts in flight so that the latest value is checkpointed. If large key-values are anticipated, Caelus can provide a special Checkpoint operations that avoid transmitting the value since the value itself does not change. Checkpointing requires all keys with conflicting values to be resolved, though data loss can of course be avoided by assigning conflicting values to new keys.

When clients intentionally leave, for example to go to sleep, they may have to delay their leaves by up to $T_S + T_A + \epsilon$ so that they can verify any operations made just prior to sleeping. Note that T_S is effectively zero in strong consistency since replication is immediate. Unfortunately, if clients have an unexpected failure that they cannot delay, a malicious server can truncate data written in the last $T_S + T_A + \epsilon$. However, a malicious server cannot omit operations since omissions will be caught by the presence checks done by other clients.

Audits. If a consistency violation is detected, either the cloud service acted maliciously or a device incorrectly reported a consistency violation. To differentiate between these two cases and enforce Guarantee CLT2, an audit procedure is required to verify that the device is truthfully reporting misbehavior by

the cloud service. The audit procedure is fairly straightforward as all the information required to perform the verification procedure is contained in the logs and attestations on the history server, and thus no information or interaction is required with the device that is accusing the cloud server. Thus, the user can perform the audit procedure by repeating the verification procedure on a device whose integrity is known to be good. We envision the requirement to do such audits to be rare, so it is reasonable to assume the user will be willing to expend some effort to acquire such a device. For example, they may boot a device from a CD or USB image that is known to be safe, or use a device capable of verified trusted-boot [27], [28]. The audit procedure can even be performed publicly if all signature verification keys are available in case the user wants to prove to a third party that the cloud service behaved maliciously.

C. Battery-friendly system

Our basic system described above is secure, but not battery-friendly because it requires the attestor to continuously produce attestations. In order to solve this problem, we introduce *attestor-partitioning*, which partitions the single attestor into a *root attestor (RA)* and an *active attestor (AA)*, each fulfilling one of the requirements on the single attestor. The device that fulfills the RA role takes on that role permanently and its identity as RA is known to all devices. However, once it selects a device to take on the role of the AA, it can sleep and conserve battery, and the selected AA will then actively create attestations every T_A. As a result, the role of the AA is not permanently attached to one device, but can be changed as necessary to minimize the impact on battery life. In Section VII, we show that Caelus has minimal battery impact on a device that is already awake – the main battery cost of Caelus results from it preventing devices from sleeping. Thus, the RA should select a device to be the AA that must be awake for other reasons – for example, the RA could select devices that the user is actively using or devices awake due to processing background tasks, such as downloading updates or synchronizing data. The RA can even select itself as the AA if it is the only device that is awake. If all devices are asleep, then no AA needs to be selected because no operations are being performed if all devices are asleep and thus, there is nothing to attest – if operations are being performed on the cloud service, then at least one device must always be awake to perform them.

While the RA can be any device, we generally envision that the user may use their smartphone as the RA for their devices. As of August 2014, there are approximately 4.6 billion cell phone subscribers worldwide and smartphones represent 65% of all new phones being sold [29]. As a result, even in instances where users only own one or a small number of devices, we can likely assume that at least one of them is a smartphone. Smartphones also have several other advantages that make them suitable for use as an RA. First, they have a cellular data connection, meaning that they are likely to be reachable and able to respond to network requests. Second, the user generally has their smartphone with them and so is more able to fix a failed or disconnected smartphone than a non-portable device. Finally, a malicious cloud provider could drop messages from the RA to make it appear that it has failed. However, since the user usually has the smartphone-RA with them, such an attack is unlikely to work as the user can easily verify the state of the smartphone.

The key security invariant that attestor-partitioning protocol must uphold is that there must not be more than one AA at any given time, otherwise a malicious cloud service can fork the AAs into two partitions that are not aware of each other, which would violate all the consistency models that Caelus tries to guarantee. Although the attestations are performed by the AA, the RA still takes an important role for keeping security guarantees by maintaining only one AA at any given time and securely handling instances where the AA fails unexpectedly.

The RA selects the AA by writing a *selection message* to the history server using a `Select` operation. After this, the AA will perform attestations every T_A, allowing the RA to sleep. Selection messages are signed and include a sequence number just like regular operations, and thus cannot be forged or replayed. They contain a timestamp and unambiguously select the client to be the AA. If the AA leaves or fails, the RA must then select another client to be the AA. Thus, while an AA is active, the RA must wake up every T_R, where $T_R \gg T_A$, to check for the presence of AA attestations in the log. If a current attestation exists, then the AA is still present and the RA renews the selection by writing a new selection message. This renewal message is important as it serves to tell the AA and clients that the AA has not been isolated from the RA. If the AA does not see a renewal at $T_R + \epsilon$ after the last renewal it must stop acting as the AA and wait until a new AA is selected.

When a client wants to join the cloud service, it checks for the presence of an AA by checking if the last selection message is less than $T_R + \delta$ old. If selection has expired, there is no current AA and the client must wake the RA. To do this, we enable clients to wake the RA by adding a `Wake` operation that causes the cloud service to wake the RA using a *push notification*. Push notifications are a facility, universally available in essentially all battery-powered devices such as phones and tablets, which allow a remote host to send a message to a mobile device, such as a RA device, and ensure it is received in a timely manner even if the device is sleeping. They utilize special hardware that puts the main processor on the device to sleep while the network interface remains awake, but allows the network interface to wake the device if a message arrives. A variety of push notification services exist, such as Google Cloud Messaging, the Apple Push Notification Service and the Amazon Simple Notification Service. Before waking the RA, the client indicates that it is awake by writing a status message to the history server using a `Status` operation. Like all other operations, status messages include a sequence number and are signed so they cannot be forged or replayed. After this, the cloud service wakes the RA, which then checks the status messages on the history server to see which devices

are awake. It then selects an active device to be AA and goes to sleep for T_R.

If the AA intends to leave, it must give up its role as the AA. Similar to the join procedure, the AA writes a status message indicating it is going to leave and asks the server to wake the RA. At that point, the RA can select a different device if there are other devices awake or go to sleep if there are no other devices awake.

D. Handling failures

One of the drawbacks of attestor-partitioning is that it can increase the likelihood of unavailability because if the AA fails, the system will become unavailable for up to $T_R + \epsilon$ for the RA to wake up, at which time the RA will detect that the AA is not making attestations. Recall that T_R could be on the order of several minutes. If other clients are awake, the RA will select a new AA, otherwise it will go to sleep. At first, it might appear that clients could avoid having to wait by waking up the RA once they detect the AA has failed (i.e. after $T_A + \epsilon$ has passed without an attestation). However, this is not safe as neither the RA nor clients can differentiate between a failed AA and a malicious cloud service who is dropping AA attestations. If the RA incorrectly assumes the AA has failed and selects a new AA when in fact the cloud service is dropping attestations, this will result in two simultaneous AAs. Without trusted communication channel between clients, the only way to avoid this is for the RA and all clients to wait until T_R has passed. After this, even if the cloud service is malicious, the AA will stop acting as an AA unless it sees a selection renewal from the RA, which the RA will not issue unless it can see the operations of the AA.

An AA can potentially suffer from a variety of failures, such as benign failures due to WiFi disconnection, battery depletion or failed hardware, as well as malicious failures such as malware infection or remote compromise. Such failed AAs affect the availability of Caelus as mentioned above. Moreover, if the failure is malicious, a compromised AA (or any other compromised client), could falsely accuse the cloud provider, requiring an audit to be run, which will impact the availability of Caelus as well. Thus, Caelus should try to minimize the chance of AA failure as much as possible. Preventing the compromise of devices by attackers is beyond the scope of this work, and we encourage readers to refer to the rich literature on intrusion detection, malware detection and system hardening. Thus, we will focus on how Caelus can minimize the chances of benign AA failures.

The easiest way to reduce benign AA failures is to have more reliable devices and networks. Enterprise-grade devices, while more expensive than consumer-grade devices, are often of higher quality, providing more reliable networks and less failure-prone hardware. Thus, while Caelus is not restricted to only enterprise settings, it will perform better in such settings where the probability of failures is low. In lieu of using higher-cost devices, Caelus can still mitigate failures by managing and configuring the system more carefully. For instance, when more than one candidate for AA is available, the RA can preferentially select an AA that is more reliable by using attributes such as previous failure history of the devices, the current battery-level of the device, network signal strength and error-rate of the device, software patch level and whether the device is in the physical presence of the user so that failures, if they do happen, can be more quickly addressed. We leave the details of an algorithm that correctly balances these and possibly other attributes for future work.

A malicious push notification service could arbitrarily delay, drop or forge notifications. Delayed or dropped notifications will reduce the responsiveness of the system since the RA will not wake up as intended. Alternatively, forged notifications will cause the RA to wake up unnecessarily, affecting battery life. Both of these attacks reduce availability, thus affecting Guarantee AV1, which is not intended to withstand malicious activity. However, all other security guarantees hold against a malicious notification service. Moreover, both of these attacks are not stealthy and can easily be detected, for example by having clients detect delayed or dropped notifications by timing the delay for the RA to respond after a `Wake`, or having the RA detect forged or delayed notifications by checking for a valid `Status` operation upon receiving a notification.

Guarantee AV1 ensures that Caelus does not increase unavailability unless the cloud service or devices fail, or there is malicious activity. The only time Caelus will stop clients from performing operations is if a scheduled attestation or selection message is missed, which can occur only when either the AA or RA fail, the network they are connected to fails or if a malicious server drops those messages.

E. Operating parameters

Caelus has a number of time-based operating parameters, some of which are dictated by the cloud service or operating environment and some of which are set by the user. T_S is the visibility time bound for the cloud service, and is a property of the distributed key-value store. We expect environmental parameters ϵ, which represents network and processing delay, to be on the order of 10s to 100s of milliseconds (for connectivity over cellular networks) and δ, which represents clock skew between devices to be a few milliseconds. Since the history server should be composed of a single machine or a set of tightly coupled machines, we expect log, attestation and select operations to take on the order of ϵ.

Caelus guarantees that clients cannot unknowingly use data that is inconsistent by more than T_{Caelus}, where $T_{Caelus} = T_S + T_A + \epsilon$. Caelus may also detect some operations that violate the shorter T_S bound, but its ability to do this is limited by how short T_A is. This is because operations logged by the history server are only attested every T_A, so an operation that violates T_S by some amount ϕ will evade detection so long as the time it waits on the history server for the next attestation cycle is less than $T_A - \phi$. Thus, a short T_A increases Caelus' ability to detect violations, but at a slightly higher network and computational cost to the AA and clients who must process attestations more often. Note that to have $T_A = 0$, which implies $T_{Caelus} = T_S + \epsilon$, this would either require

an AA that checks infinitely often or a trusted history server that implements the AA.

While a RA that is never unavailable can never cause the system to be unavailable, real devices do become unavailable so they can cause system unavailability. Attestor-partitioning mitigates the effects of RA unavailability by allowing the AA to hide some of the times the RA is unavailable. While the value of T_R does not affect system security, a longer T_R reduces the likelihood that temporary unavailability of the RA will affect unavailability of the entire system. To illustrate, consider a RA running on the smartphone with an availability of 97.81% as found by our informal measurement study detailed in the Appendix. Modelling the phone as a random variable with an expected value of 97.81% and is subject to a trial every T_R, it would take about $32 \times T_R$ before the probability that phone unavailability impacts Caelus' availability increases to 50%. Considering an average period of unavailability of approximately 94 seconds (again from the study), partitioning gives the user a 50% chance of experiencing roughly 7 minutes of unavailability every 24 hours with a T_R of 10 minutes, which compares favorably to the expected 30 minutes of unavailability the same phone is expected to experience every 24 hours. Moreover, the expected unavailability decreases exponentially as phone availability increases. Therefore, we think the smartphone RA can be reliable and highly available for our target environment.

In general, a longer T_R will also improve battery life of the RA as the device the RA is on can spend a longer proportion of time sleeping. Mobile push notification services typically require the device to wake up and send keep-alive messages every 5-10 minutes. As a result, we generally expect that T_R is set to coincide with these keep-alive periods.

V. Security Analysis

Now that we have presented the design of Caelus we describe how the individual guarantees that Caelus provides are upheld by elements in its design. These guarantees hold when either the cloud service is malicious or clients are malicious. We further show that even if several malicious clients collude, they do not gain any abilities beyond those that a single malicious client has.

A. Analysis of guarantees

SRV1. Since all clients encrypt data before sending it to the cloud provider and the encryption key is not known to the cloud service, a malicious cloud provider cannot read user data.

SRV2. A cryptographic hash of data sent in `Puts` is computed and included in the header of the `Put`. The header is signed by the device making the update. The same header is returned by the cloud service to a client performing a `Get` on the same key. The client then verifies the signature on the header and then uses the hash to verify the integrity of the returned data. Since the cloud service cannot forge the signatures, the data stored on the cloud is protected from tampering by a malicious cloud provider.

SRV3. The scheduled attestations produced by the AA in combination with the consistency model-specific verification checks guarantee that consistency violations are detected within T_{Caelus}. Scheduled attestations ensure that all clients are notified of the history of an operation within T_{Caelus} after the operation occurs. Since all clients see the same history, one can view the attested history as a "global history" of all operations. The verification checks then guarantee two properties: 1) each client's observed history of operations matches the attested global history and 2) the global history is consistent with the promised consistency model. For the second property, the verification checks verify that all operations are made visible within the promised time bound by comparing the timestamps on operations (i.e., no stale data is read) and that operations are made visible according to the ordering constraints specified by the promised consistency model (i.e., no malicious reordering).

CLT1. Because all key-value updates must be signed by the client making them, a client cannot later deny that it made the update. As a result, data modifications are non-repudiable.

CLT2. To falsely accuse the cloud service of a consistency violation, a client must show that one of the verification checks has failed even when in reality it hasn't. This can only happen in one of two ways: either the accusing client is able to alter the contents of the attested history so that a check fails, or it can convince the user that a verification check has failed even when it hasn't.

All attested history segments are signed twice, once initially by the cloud service and then again by the attestor. As a result, a regular client would have to forge the signatures of both the cloud service and the attestor to tamper with the attested history. If either the RA or the AA is malicious, it could try to tamper with the attested history before signing it. However, to successfully tamper with it, the malicious attestor would still need to forge the cloud service's signature, which is not possible according to our attack model. As a result, no malicious client can tamper with the attested history to falsely accuse an honest cloud service.

The other alternative is for the client to incorrectly evaluate a verification check and declare that a check has failed. However, since all consistency violations can be publicly audited, a malicious client on its own cannot falsely accuse the cloud service.

AV1. Clients expect the AA to sign an attestation every T_A. If this does not happen, then clients will halt, affecting the availability of the system. Similarly, a malicious RA can refuse to select AA and also refuse to sign attestations. If the cloud service is malicious, it can also affect availability by simply refusing to respond to requests for clients. Thus a single malicious client, if it happens to be the AA or RA, or a malicious cloud service can invalidate Guarantee AV1.

Under normal circumstances, where there are no failures and all components adhere to the protocol, an attestation is produced every T_A. If all clients are asleep and a new client joins, it must wake up the RA. Normally, the new client would

have to wait for up to T_R for the RA to wake up, thus affecting availability. However, because Caelus uses push notifications, this waiting period is shortened to the latency of a push notification, which is on the order of 1 second. Thus, under normal circumstances, Caelus does not affect availability.

B. Colluding clients

Multiple colluding clients do not have any capabilities beyond a single malicious client. Like a single client, they are only able to corrupt or leak data by virtue of their ability to access and modify the data. However, both guarantees against malicious clients, CLT1 and CLT2, hold.

CLT1. Malicious clients could share their signing keys thus allowing any malicious client to forge signatures that could have been made by another malicious client. Thus, malicious actions can no longer be traced to the client that made them, but only to a member of the group of colluding malicious clients. While this changes the actual terms of Guarantee CLT1, it doesn't change the intent – actions made by an adversary in control of several clients are still traceable back to that group of clients. Thus, Guarantee CLT1 still holds in the face of client collusion.

CLT2. To falsely accuse the cloud service, clients must be able to forge cloud service signatures. Having more than one malicious client does not make it any more possible to forge signatures, so Guarantee CLT2 will also hold against colluding clients.

In summary, colluding clients, regardless of whether they are regular clients or include the AA or RA, do not invalidate any guarantees except Guarantee AV1, which can already be invalidated by a single malicious client if that client happens to be an AA or RA.

VI. Implementation

In this section, we describe our Caelus prototype, which implements the cloud server and history server components in the cloud service and clients for PCs and Android devices.

A. Cloud service

The cloud server and history server components are implemented in 3K lines of Java and communication between the server components as well as clients is implemented using Apache XML-RPC. To implement different consistency models, our prototype is modular and can use different key-value store backends. For strong consistency, our prototype uses a single cloud server with a local key-value store implemented with the LevelDB library [30]. Because all clients communicate with a single server, the server can make all client operations atomic, thus providing strong consistency. To implement eventual consistency, our prototype uses Amazon's cloud infrastructure. Multiple cloud servers run as EC2 instances and use Amazon's S3 service as the key-value store backend. A single history server typically shares one of the EC2 nodes with a cloud server but could also run on a dedicated node. We are currently not aware of any open-source or commercial cloud service that implements a causally consistency key-value store. As a result, our prototype doesn't implement causal consistency at this time. However, if one were available, we believe it would be fairly straight forward to integrate it with our prototype as our prototype only assumes a `Put` and `Get` interface.

It is possible to implement Caelus without cloud server components by having clients communicate directly with the key-value store and history server. However, having the cloud server allowed us to easily abstract the different LevelDB and S3 interfaces from the clients, allowing us to have identical client code for all experiments.

B. Clients

We implement two different types of clients for our prototype, one for PCs and one for Android devices. Both are written in Java and consist of about 7K lines of code. To reduce the number of client-server round trips the server piggy backs recent attestations on the responses to `Put`s and `Get`s.

Each time a client performs a `Put`, it is enqueued on a *deferred verification list*. Occasionally, a `Get` can be verified at the time it occurs because it reads the latest attested value, but other times, it reads a value that has yet to be attested, so it must also be enqueued on the deferred verification list. Verification of deferred operations is performed asynchronously by a *verification thread*, which periodically wakes up every T_A, processes any new log segments that have been received and then verifies operations on the deferred verification list. Operations that remain unverified for longer than T_{Caelus} are flagged as violations. Any delays between when the AA posts attestations and when clients process them must be accounted for in ϵ. Thus, we synchronize both the period and phase of the verification thread with that of the AA.

We use the Google Cloud Messaging (GCM) service to implement push messages on Android clients. GCM generally takes about 1 second to deliver a message to the phone because it requires an additional network hop to Google's servers. This latency could be reduced by implementing our own dedicated push service and collocating the notification server with the cloud server the phone is connected to, but for the purposes of our prototype, we found the 1 second latency to be reasonable and perhaps more realistic since most cloud services would more likely use a third-party notification service than implement their own. GCM does not use a fixed period for keep-alive messages, but instead varies their timing depending on network conditions. Caelus can be modified to allow for a variable T_R by having the phone embed the length of the current AA selection period in each selection message, but our prototype does not implement this. As a result, we currently do not synchronize T_R with the GCM heartbeat period.

VII. Evaluation

We evaluate four properties of our Caelus prototype. First, we evaluate Caelus' effectiveness at detecting consistency violations. Second, we evaluate the computational costs of Caelus on clients. Third, we evaluate the battery costs of Caelus on

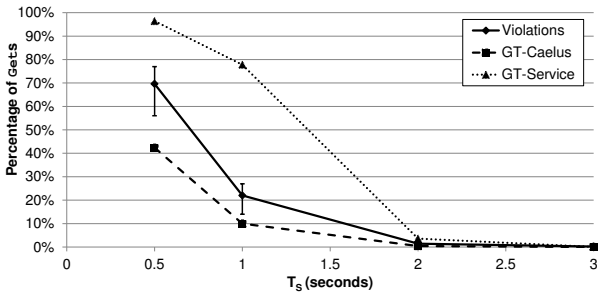

Fig. 5. Percentage of Gets with consistency violations on S3 as a function of T_S. GT-Caelus represents the true number of T_{Caelus} violations and GT-Service represents the true number of T_S violations.

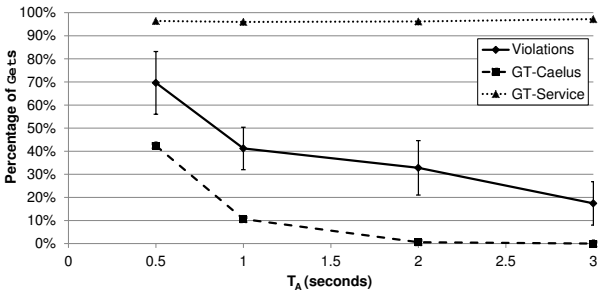

Fig. 6. Percentage of Gets with consistency violations on S3 as a function of T_A.

the smartphone, as well as the battery savings of attestor-partitioning. Finally, we evaluate the network bandwidth overhead of sending and retrieving attestations in Caelus.

A. Detecting consistency violations

We begin by evaluating Caelus' effectiveness at detecting consistency violations using our eventual consistency prototype on S3. Amazon does not publish a visibility time bound for S3. Thus, we vary T_S and measure the effect on the number of consistency violations detected by Caelus. Using a T_S smaller than what S3 supports simulates a malicious cloud provider who tries to claims a shorter visibility time bound than what they can deliver.

We deploy Caelus on S3 in the US Standard Region, which automatically replicates data across Amazon data centers in the USA. We then deploy cloud servers on EC2 in the Oregon data center on the west coast using a t1.micro instance with 2GHz Intel Xeon E5-2650 cores and 600MB memory and the Northern Virginia data center on the east coast using a m3.2xlarge instance with an 8-core 2.5GHz Intel Xeon E5-2670 Processor and 30GB of memory. Four "writer" clients are running on the Northern Virginia server and repeatedly perform Puts of non-repeating 1MB values on a key. A "reader" client is connected to the Oregon server and repeatedly performs Gets on the same key. The reader client runs on the smaller t1.micro instance with 2GHz Intel Xeon E5-2650 cores and 600MB of memory. We set ϵ to be 100ms, δ to be 5ms, $T_A = 500ms$ and vary T_S between 0.5 and 3 seconds, taking the average over 5 runs. We also log the time of every Put and Get and perform an offline analysis to extract the ground truth (GT) number of T_{Caelus} and T_S violations. We then plot the results in Figure 5. As stated by its guarantees, Caelus detects all T_{Caelus} violations and some but not all T_S violations. As the T_S increases, more operations are replicated by S3 in time, resulting in fewer true and detected violations.

In Figure 6, we hold T_S fixed at 0.5 seconds while varying T_A between 0.5 and 3 seconds. The number of true violations of T_S stays the same, but the number of true T_{Caelus} violations and those detected by Caelus decreases as T_A increases, illustrating how a larger T_A decreases Caelus' ability to detect T_S violations.

TABLE I
CONSISTENCY VERIFICATION PERFORMANCE ON A PC.

	Attest (μs)	Presence (μs)	Consistency (μs)
Strong-Get	85.7±2.02	399.25±11.64	86.1±13.21
Strong-Put	85.7±2.02	402.38±26.08	9.52±2.19
Eventual-Get	67.5±2.26	392.59±12.57	17.79±2.71
Eventual-Put	67.5±2.26	410.78±8.45	18.47±6.68
Causal-Get	97.6±2.88	307.44±16	595.69±50.43
Causal-Put	97.6±2.88	319.03±8.15	2.7±1.75

B. Client verification costs

Since Caelus verification operations occur asynchronously, they are not on the critical path of any Put or Get operations and thus do not affect the performance of these operations. However, Caelus does increase CPU utilization as both verification and attestation contain cryptographic (2048 bit RSA with SHA256) and logical computations. We evaluate the computational costs of the different consistency model verification procedures by running them against our strong consistency prototype. The strong consistency server never causes consistency violations and evaluates the worst case computational costs because operations must pass all tests to verify correctly while Caelus will not perform any further checks on an operation once it detects that an operation violates consistency.

We measure the time to perform verifications on both a PC with 3.4GHz Intel i7-2600 Processor and 16GB of memory and on a rooted stock Google Nexus 5 phone with 2.3GHz processor and a 2300 mAh battery. We run measurements in our lab to minimize network variability, and therefore machines are connected to a local Caelus service also in our lab. We run YCSB [31] with a 50/50 mix of Put and Get operations with no delay between operations on both machines, resulting in an applied workload of 26 ops/s. T_A

TABLE II
CONSISTENCY VERIFICATION PERFORMANCE ON A SMARTPHONE.

	Attest (ms)	Presence (ms)	Consistency (ms)
Strong-Get	1.49±0.06	2.24±0.33	0.05±0.03
Strong-Put	1.49±0.06	2.15±0.14	0.01±0.01
Eventual-Get	1.40±0.12	1.91±0.11	0.74±0.22
Eventual-Put	1.40±0.12	2.22±0.11	0.03±0.01
Causal-Get	1.75±0.13	1.79±0.13	2.53±0.39
Causal-Put	1.75±0.13	2.18±0.09	0.022±0.01

TABLE III
BATTERY SAVINGS AND PERCENTAGE TIME SLEEPING COMPARISON BETWEEN WHEN PHONE ACTS AS AN ATTESTOR AND WHEN ATTESTOR-PARTITIONING IS USED.

	Battery (mAh)	Sleeping (%)
Idle	20.85	98.5
Single Attestor (WiFi)	90.2	0
Single Attestor (LTE)	90.29	0
Root Attestor (WiFi)	22.57	98.3
Root Attestor (LTE)	22.57	97.7

is set to 1 second and we take the average over 5 runs.

The per-operation cost of the individual steps in the verification procedure are tabulated for the PC in Table I and for the Nexus 5 in Table II. The consistency column records the cost of all the model-specific consistency checks, which are generally fast with the exception of `Gets` under causal consistency. This check requires an iterative search through the log to find all operations with vector clocks between the `Get` and matching `Put`. Cryptographic operations are main source of overhead for Caelus. Out of the three components of the verification operation, the presence check component dominates the overall cost because there is a public-key signature verification performed on each operation in the log segment. These relative trends hold on both the PC and the Nexus 5, except that the PC is roughly 5-18× faster at cryptographic operations, which is to be expected. We also evaluate the cost of performing the signing operations and attestations and found that they are dominated by the cost of the RSA signature operation, which takes about 11ms on the PC and 60ms on the Nexus 5 regardless of the type of operation being signed.

Overall, the cost of Caelus operations is not high and we find that these operations take about 8.8-16.4% of CPU time on our test devices. Currently, our Caelus prototype signs and verifies individual cloud operations and this makes up the bulk of the CPU overhead. Batching signing cloud operations would reduce both the number of signatures and verifications and thus reduce the CPU overhead of Caelus.

C. Phone battery consumption

When regular Caelus client devices have no operations to perform on the cloud service, they can perform a client leave and go to sleep, so Caelus imposes no battery cost on normal client devices. The only devices that have additional duties in Caelus even if they have no operations to perform is the RA and the AA. We thus measure the battery impact on the RA when it could be otherwise idle. In addition, recall that the RA should select an AA that is already awake, so we also measure the battery impact of Caelus on an AA that is running other tasks.

We used the same phone we used for verification cost measurement. We use battery level readings from the OS and the percentage of time the phone spends sleeping to measure the benefits of attestor-partitioning. To get a baseline, we first perform measurements on an idle phone in its default configuration with basic services and applications running and background synchronization disabled. We then compare this to the battery consumption of the phone acting as a single attestor in the basic system and Root Attestor using attestor-partitioning. For these experiments, we have clients run a simulated image browsing and editing workload with a mix of random 330 `Gets` and 30 `Puts` of 1MB values every 30 minutes. T_A is set to 1 second and T_R is set to 5 min. We perform measurements when the phone is on a WiFi network and a cellular LTE network. We run each experiment for at least 30 minutes, or longer until workload is finished, and normalize the results to a 30 minute period in Table III. Our battery consumption measurement tool rounds up to the nearest percentage of battery capacity (i.e. 23 mAh). The results show that attestor-partitioning has negligible battery use over a completely idle phone with only a slight increase in battery usage and a slight decrease in time spent sleeping. However, compared to the phone acting as a single attestor, running as an RA with attestor-partitioning reduces the additional battery drain over an idle phone by about 40×.

TABLE IV
BATTERY DRAIN AND AVERAGE CPU FREQUENCY OF AN ACTIVE PHONE WITH AND WITHOUT THE ATTESTOR ROLE.

	Battery (mAh)	CPU (GHz)
Active Attestor (WiFi)	431.01	1.66
Active Attestor (LTE)	433.87	1.52
No Caelus (WiFi)	366.17	1.64
No Caelus (LTE)	343.66	1.61

TABLE V
NETWORK BANDWIDTH CONSUMED BY CAELUS OPERATIONS.

Operation	Cost (Bytes)
`Read_History`	$1411 + 1087 \times \|Puts\| + 695 \times \|Gets\|$
`Write_Attest`	756
`Read_Attest`	$2582 + 1087 \times \|Puts\| + 695 \times \|Gets\|$
`Select`	1421

In cases where a device is acting as an AA, the device is assumed to be running some other workload that prevents it from sleeping. To evaluate the battery impact on such a device, we run the same image viewing workload as above on the phone. To simulate UI events, we use the *monkey* tool, which generates random UI events. We compare the battery drain and average CPU frequency when the phone is acting as an Active Attestor with when Caelus on the phone is completely disabled and tabulate the results in Table IV. To isolate the cost of Caelus components, fetching and verifying attestations is disabled in all "No Caelus" cases. The results show that acting as an attestor on an active phone adds roughly a 17-26% increase in battery consumption.

D. Network cost

On top of existing `Get` and `Put` operations, Caelus adds operations that fetch log segments, read and write attestations, as well as assigning the attestor role through select operation – all of which consume network bandwidth. To measure this cost, we measure the amount of data before it is encoded by XML-RPC. Because XML-RPC transmits data in ASCII, it Base64 encodes encrypted binary data, which adds about 1.5× overhead. Using a binary packet format in our prototype

would have avoided this unnecessary artifact. We note that this measurement method also doesn't take into account transport protocol overhead, but these costs are well understood (usually about 40-60 bytes per packet).

Table V gives the cost of various Caelus control operations. Note that the cost of `Read_History` and `Write_History` operations depend on the number of `Put` and `Get` operations that are attested as this affects the size of the history log segment that is read. For the image browsing workload in the previous section, the client uses about 1.14MB on `Read_Attest` messages, the AA uses about 3.65MB on `Read_History` and `Write_Attest` messages and the phone uses about 8.33KB on `selection` messages. When amortized over the 360MB of data transferred in the workload, this works out to about 13KB of network bandwidth overhead per megabyte of transferred data or about 1.3%. While these costs are fairly small, they are actually smaller in practice since they only exist if clients are active and using the cloud service. If the cloud service is not being used, the clients use no network bandwidth at all.

VIII. Related Work

The most closely related works to Caelus are SUNDR [3], BFT2F [8], and CloudProof [9]. All of these systems provide consistency and integrity guarantees for untrusted storage systems to clients who do not communicate directly with each other. SUNDR only guarantees fork consistency, while BFT2F weakens fork consistency to fork*. Other work has also extending SUNDR's contribution on fork-linearizability to computations on untrusted services [32], [33]. Both fork, fork* and fork-linearizability are weaker than any of the consistency models that Caelus can guarantee in that they permit some operations to be forever unknown to some clients. CloudProof can verify strong consistency, but requires information from clients to be assembled at an "auditor". Because the auditor is not always online, auditing is retroactive instead of in real time. Caelus uses a smartphone to make auditing real-time and distributes the auditing work to minimize the impact on the smartphone battery.

Depot [4], SPORC [5] and Venus [6] provide consistency guarantees using client-to-client communication. Client-to-client communication simplifies the problem because clients may implement their own replication policy and thus enforce a consistency model independent of the cloud service provider. However, client-to-client communication is either inefficient for battery-powered devices, or it requires a trusted service that can buffer and multicast messages so that clients need not waste battery on network bandwidth or need not be simultaneously awake to communicate. Caelus avoids these by devising a protocol that can use the cloud service to buffer and multicast messages without having to trust the cloud service.

Timeweave [34] was an early use of attested histories to verify the actions of an untrusted party. Since then, the idea of using an attested history has been applied to detect misbehaving virtual machines [35], misbehaving replicas in BFT systems [36], [37], as well as to improve the performance of BFT systems [38], [39]. Recent work has also proposed the use of trusted platform modules (TPMs) as integrity verifiers for cloud infrastructure [40]. However, none of these previous works directly address the problem of consistency verification.

Several cryptographic file systems also guarantee freshness [41]–[46]. However, they all assume that all operations are linearizable so that the only need to check that the latest values are read by a client (i.e. strong consistency). As stated in the Brewer's well-known CAP theorem, systems that enforce strong consistency cannot scale. In contrast, Caelus provides protection on systems with weaker consistency models such as eventual and causal consistency, which are more suitable for globally distributed cloud infrastructure.

Finally, other work has proposed distributing data cross multiple cloud services to protect the integrity and recoverability of data [47]–[49], as well as using cryptographic techniques to probabilistically prove retrievability [50], [51], data possession [52], or whether data is encrypted properly [53]. However, these systems do not address the consistency of data and in the case of the cryptographic techniques, mostly assume static data. On the other hand, Caelus does not directly address recoverability or retrievability, making some combination of these techniques with Caelus interesting future work.

IX. Conclusion

From designing and evaluating our Caelus prototype, we draw two major conclusions. First, Caelus is able to avoid direct client-to-client communication and use the untrusted cloud provider for communication by offloading parts of the monitoring task in a way that doesn't require trust in the cloud service provider. Second, the role of the attestor can be partitioned into two root and active attestor devices that each fulfill one of the roles required of the single monolithic attestor. Through careful protocol design, Caelus ensure that the root and active attestor devices cannot be partitioned by a malicious cloud service.

Our evaluation shows that Caelus is able to detect consistency violations on Amazon's S3 storage service in a compute and battery efficient manner. Attestation-partitioning reduces the battery impact of the root attestor by about 40× and the cost in CPU time and network bandwidth overhead is minimal.

Acknowledgement

We thank our shepherd Alina Oprea, for her wonderful guidance. We also thank Wei Huang, Afshar Ganjali, Sukwon Oh, Ding Yuan, Michael Stumm, Ashvin Goel, Eyal de Lara and Angela Demke-Brown for their helpful comments. This research was supported by an ORF-RE grant from the Ontario Ministry of Research and Innovation and by an NSERC Discovery Grant.

References

[1] J. Cook, "Google Drive Now Has 10 Million Users: Available on iOS and Chrome OS," http://techcrunch.com/2012/06/28/google-drive-now-has-10-million-users-available-on-ios-and-chrome-os-offline-editing-in-docs/.

[2] P. Maass and L. Poitras, "Core Secrets: NSA Saboteurs in China and Germany," https://firstlook.org/theintercept/2014/10/10/core-secrets.

[3] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure Untrusted Data repository (SUNDR)," in *The 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[4] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud Storage with Minimal Trust," in *The 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[5] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group Collaboration using Untrusted Cloud Resources," in *The 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[6] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, "Venus: Verification for Untrusted Cloud Storage," in *The 2010 ACM Workshop on Cloud Computing Security Workshop (CCSW)*, Oct. 2010.

[7] B. H. Kim, W. Huang, and D. Lie, "Unity: Secure and Durable Personal Cloud Storage," in *The 2012 ACM Workshop on Cloud Computing Security Workshop (CCSW)*, Oct. 2012.

[8] J. Li and D. Mazieres, "Beyond one-third faulty replicas in byzantine fault tolerant systems." in *The 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2007.

[9] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang, "Enabling Security in Cloud Storage SLAs with CloudProof," in *The 2011 USENIX Annual Technical Conference (ATC)*, Jun. 2011.

[10] T. Lodderstedt and M. Scurtescu, "OAuth 2.0 Token Revocation," IETF RFC 7009, Aug. 2013.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-Distributed Database," in *The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2012.

[12] K. Fall, "A Delay-Tolerant Network Architecture for Challenged Internets," in *The 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, Aug. 2003.

[13] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically Bounded Staleness for Practical Partial Quorums," *The VLDB Endowment*, vol. 5, no. 8, pp. 776–787, Apr. 2012.

[14] W. Golab, M. Rahman, A. Auyoung, K. Keeton, and I. Gupta, "Client-Centric Benchmarking of Eventual Consistency for Cloud Storage Systems," in *The 34th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2014.

[15] D. Bermbach and S. Tai, "Eventual Consistency: How Soon Is Eventual? An Evaluation of Amazon S3's Consistency Behavior," in *The 6th Workshop on Middleware for Service Oriented Computing (MW4SOC)*, Dec. 2011.

[16] E. Anderson, X. Li, M. A. Shah, J. Tucek, and J. J. Wylie, "What Consistency Does Your Key-Value Store Actually Provide?" in *The 6th International Conference on Hot Topics in System Dependability (HotDep)*, Oct. 2010.

[17] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie, "Toward a Principled Framework for Benchmarking Consistency," in *The 8th USENIX Conference on Hot Topics in System Dependability (HotDep)*, Oct. 2012.

[18] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data Consistency Properties and the Tradeoffs in Commercial Cloud Storages: the Consumers' Perspective," in *The 5th Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2011.

[19] S. Patil, M. Polte, K. Ren, W. Tantisiriroj, L. Xiao, J. López, G. Gibson, A. Fuchs, and B. Rinaldi, "YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores," in *The 2nd ACM Symposium on Cloud Computing (SOCC)*, Oct. 2011.

[20] D. Bermbach, S. Sakr, and L. Zhao, "Towards Comprehensive Measurement of Consistency Guarantees for Cloud-Hosted Data Storage Services," in *The 5th TPC Technology Conference on Performance Evaluation & Benchmarking (TPCTC 2013)*, Aug. 2013.

[21] W. Santos, "76 Storage APIs: Box.net, Amazon S3, Dropbox," http://www.programmableweb.com/news/76-storage-apis-box.net-amazon-s3-dropbox/2012/01/31.

[22] B. Voo, "20+ Tools To Supercharge Your Dropbox," http://www.hongkiat.com/blog/dropbox-tools/.

[23] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding Personal Cloud Storage Services," in *The 2012 ACM conference on Internet measurement conference (IMC)*, Nov. 2012.

[24] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-Based Service Level Agreements for Cloud Storage," in *The 24rd ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS," in *The 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2011.

[26] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability Using Lazy Replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 360–391, Nov. 1992.

[27] M. Mannan, B. H. Kim, A. Ganjali, and D. Lie, "Unicorn: Two-Factor Attestation for Data Security," in *The 18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011.

[28] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An Execution Infrastructure for TCB Minimization," *SIGOPS Opering Systems Review*, vol. 42, no. 4, pp. 315–328, Apr. 2008.

[29] Ericsson Mobility, "Interim Update: Ericsson Mobility Report," http://www.ericsson.com/res/docs/2014/ericsson-mobility-report-august-2014-interim.pdf.

[30] S. Ghemawat and J. Dean, "Leveldb: A Fast and Lightweight Key/Value Database Library by Google," https://code.google.com/p/leveldb/.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *The 1st ACM Symposium on Cloud Computing (SOCC)*. ACM, 2010, pp. 143–154.

[32] C. Cachin, "Integrity and Consistency for Untrusted Services," in *The 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, Jan. 2011.

[33] C. Cachin and O. Ohrimenko, "Verifying the Consistency of Remote Untrusted Services with Commutative Operations," in *The 18th International Conference on Principles of Distributed Systems (OPODIS)*, Dec. 2014.

[34] P. Maniatis and M. Baker, "Secure History Preservation through Timeline Entanglement," in *The 11th USENIX Security Symposium*, Aug. 2002.

[35] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable Virtual Machines." in *The 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[36] A. Haeberlen, P. Kouznetsov, and P. Druschel, "PeerReview: Practical Accountability for Distributed Systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 175–188, Dec. 2007.

[37] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," in *The 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[38] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested Append-Only Memory: Making Adversaries Stick to their Word," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 189–204, Dec. 2007.

[39] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small Trusted Hardware for Large Distributed Systems," in *The 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2009.

[40] J. Schiffman, Y. Sun, H. Vijayakumar, and T. Jaeger, "Cloud Verifier: Verifiable Auditing Service for IaaS Clouds," in *2013 IEEE Ninth World Congress on Services (SERVICES)*, Jun. 2013.

[41] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea, "Iris: A Scalable Cloud File System with Efficient Integrity Checks," in *The 28th Annual Computer Security Applications Conference (ACSAC)*, Dec. 2012.

[42] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing Remote Untrusted Storage," in *The 10th Symposium on Network and Distributed System Security (NDSS)*, Feb. 2003.

[43] R. L. Rivest, K. Fu, and K. E. Fu, "Group Sharing and Random Access in Cryptographic Storage File Systems," Masters Thesis, MIT, Tech. Rep., Jun. 1999.

[44] M. T. Goodrich, C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Athos: Efficient Authentication of Outsourced File Systems," in *The 11th International Conference on Information Security (ISC)*, Sep. 2008.

[45] A. Barsoum and A. Hasan, "Enabling Dynamic Data and Indirect Mutual Trust for Cloud Computing Storage Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 12, pp. 2375–2385, Dec 2013.
[46] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolic, "PoWerStore: Proofs of Writing for Efficient and Robust Storage," in *The 20th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2013.
[47] M. Vrable, S. Savage, and G. M. Voelker, "BlueSky: A Cloud-Backed File System for the Enterprise," in *The 10th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2012.
[48] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds," *ACM Transactions on Storage (TOS)*, vol. 9, no. 4, p. 12, Nov. 2013.
[49] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: A High-Availability and Integrity Layer for Cloud Storage," in *The 16th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.
[50] A. Juels and B. S. K. Jr., "PORs: Proofs of Retrievability for Large Files," in *The 14th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2007.
[51] H. Chen and P. Lee, "Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage: Theory and Implementation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 2, pp. 407–416, Feb 2014.
[52] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable Data Possession at Untrusted Stores," in *The 14th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2007.
[53] M. van Dijk, A. Juels, A. Oprea, R. L. Rivest, E. Stefanov, and N. Triandopoulos, "Hourglass Schemes: How to Prove that Cloud Files Are Encrypted," in *The 19th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2012.

APPENDIX

[Smartphone Connectivity Study]

While smartphones are designed to be constantly connected and cellular coverage is available in most populated areas of the world, momentary gaps in cellular connectivity is still quite a common occurrence. To better understand this phenomenon, we performed an informal smartphone connectivity study. We acknowledge that our study has limitations – the participants are from the same geographical area so the study is limited to the 4-5 carriers who service the area. However, given that cellular coverage will only continue to improve in all parts of the world, we believe that the results we attain here should be representative of what most populated areas of the world will be able to achieve in the near future.

To record the availability of phones, we built a simple Android application that records the periods when the phone is not connected to the Internet. The application continually monitors network connectivity on both cellular and WiFi interfaces by registering for network status events. The application was installed on the phones of 12 participants over a 7 month period. At the end of the period, we measured the total time the phones had network connectivity over the total monitored time to be 97.81%. The average duration of a disconnection was roughly exponentially distributed, with a mean of 94 seconds and a longest measured period of disconnection to be 5.7 hours. About 90% of disconnections last for less than 2 minutes suggesting that even if smartphone unavailability is encountered, it wouldn't last long enough for a human user to perceive too much inconvenience. In addition, we found that most disconnection events tended to be clustered, suggesting they were related to the user's physical location. Thus, if the user is trying to access the cloud service while in an area of poor reception, they can likely remedy the situation by moving to a different location.

Various industry measures indicate that smartphone usage is rising, so intuitively one would believe that cellular networks have to be fairly reliable to have fostered such heavy use. Our findings do not contradict this intuition and they suggest that smartphones do indeed have a high enough level of connectivity that episodes of connectivity loss are short and isolated.