Don’t Trust the Cloud, Verify: Integrity and Consistency for Cloud Object Stores

Marcus Brandenburger Christian Cachin Nikola Knežević
IBM Research - Zurich
CH-8803 Rüschlikon, Switzerland
(bur|cca|kne)@zurich.ibm.com
11 February 2015

Abstract
Cloud services have turned remote computation into a commodity and enable convenient online collaboration. However, they require that clients fully trust the service provider in terms of confidentiality, integrity, and availability. Towards reducing this dependency, this paper introduces a protocol for verification of integrity and consistency for cloud object storage (VICOS), which enables a group of mutually trusting clients to detect data-integrity and consistency violations for cloud storage. It aims at services where multiple clients cooperate on data stored remotely on a potentially misbehaving service. VICOS enforces the consistency notion of fork-linearizability, supports wait-free client semantics for most operations, and reduces the computation and communication overhead compared to previous protocols. VICOS is based in a generic way on any authenticated data structure. A prototype of VICOS that works with the key-value store interface of commodity cloud storage has been implemented, and an evaluation demonstrates its advantage compared to existing systems.

1 Introduction
More and more people outsource their data to the cloud, and collaborating on a shared resource using cloud services has become easier than ever. Programmers work together on online source code repositories, global project teams produce technical deliverables, and friends share their photo albums. Nevertheless, the clients need to trust the cloud provider and rely on it for the confidentiality and correctness of their data. Encryption may preserve the confidentiality of data but cannot prevent inadvertent or malicious data modifications. This work shows how to protect data integrity and consistency for an untrusted cloud storage service accessed by multiple clients.

With a single client only, the client may locally keep a short cryptographic hash value of the outsourced data. Later, this can be used to verify the integrity of the data returned by the cloud storage service. However, with multiple (disconnected) clients, no common synchronization, and no communication among the clients, neither hashing nor digital signatures are sufficient by themselves. The reason is that a malicious (or Byzantine) server may violate the consistency of the data, for example, by reordering or omitting properly authenticated operations, so that the views of the storage state at different clients diverge. A malicious cloud server may pretend to one set of clients that some operations by others simply did not occur. In other words, freshness can be violated and the clients cannot detect such attacks until they communicate directly. The problem is particularly relevant for cryptographic online voting and for web certificate transparency [18].
The strongest achievable notion of consistency in this multi-client model is captured by \textit{fork-line-arizability}, introduced by Mazières and Shasha [22]. A consistency and integrity verification protocol may guarantee this notion by adding condensed data about the causal evolution of the client’s views into their interaction with the server. This ensures that if the server creates only a single discrepancy between the views of two clients, these clients may never observe operations of each other afterwards. In other words, if the server ever lies to some clients and these clients communicate later, they will immediately discover the violation. Hence, with only one check they can verify a large number of past operations.

The SUNDR system [19] pioneered fork-linearizable consistency and demonstrated a network file system protected by a hash tree of every user’s files. SUNDR uses an expensive protocol, requiring messages of size $\Omega(n^2)$ for $n$ clients [9]. Like other systems providing fork-linearizability, it suffers from the inherent limitation that sometimes, even with a correct server, clients have to block and cannot proceed with their next operation because other clients are concurrently executing operations [9].

In order to prevent blocking, FAUST [7], SPORC [12], and Venus [25] relax their guarantees to \textit{weak fork-linearizability}, which establishes consistency only eventually, after further operations occur; this is not desirable because a client may only know later that a protocol output was not correct. SPORC and the related Blind Stone Tablet (BST) [28] protocol shift the maintenance of state to the clients, such that the server is merely responsible for coordination; every client holds a complete copy of the system’s state. This contradicts the goal of outsourcing data to the cloud. BST and COP [8] circumvent blocking for all operations that commute with each other.

In this paper, we present VICOS, a \textit{verification protocol for the integrity and consistency of cloud object storage}, which overcomes these limitations. VICOS supports the optimal consistency notion of fork-linearizability, provides wait-free semantics for all compatible client operations, and has smaller overhead than previous protocols. Ensuring progress for all compatible operations, as defined here, is more general than considering only commuting operations as in past work. Conceptually, VICOS is based on abstract \textit{authenticated data structures} and unifies the two different lines of work on this problem so far, namely, the untrusted storage protocols [19, 9, 7] that feature remote state and are based on vector clocks, and the remote service verification protocols [28, 12, 8], which create local copies of the state and use hash chains. Furthermore, VICOS incurs only a constant communication overhead per operation, independent of the number of clients, whereas FAUST and Venus require vector clocks of size $\Omega(n)$.

We have implemented and evaluated VICOS with a commodity cloud object store; the results demonstrate that the overhead for distributed, multi-client integrity and consistency verification is low. VICOS protects the complete cloud storage service, spanning many objects and offering read, write, delete, and directory listing operations; this stands in contrast to Venus [25], which only provided consistency for a single data object. Furthermore, the protocol notifies the clients whenever the integrity or consistency is violated but does not address recovery operations.

\subsection*{1.1 Contributions}

This work makes the following contributions towards ensuring integrity for data stored on untrusted cloud providers:

- A novel abstract protocol to verify the integrity and consistency of a generic service based on an authenticated data structure [27], which ensures fork-linearizability, supports wait-free semantics for compatible operations, and incurs only constant communication overhead.

- An instantiation of this protocol for a commodity cloud object store, VICOS. It represents the first integrity protection protocol with all of the above features for the standard operations of a cloud object store.
• An implementation and evaluation of VICOS using COSBench, demonstrating its practicality. In particular, the overhead of VICOS for integrity protection remains unnoticeably small with moderate concurrency and increases slightly when many clients access the same data concurrently.

1.2 Organization

The paper continues with introducing the model and defines our notion of an authenticated data structure (ADS). Sec. 3 presents the abstract integrity protocol for ADS and discusses its properties. VICOS is introduced in Sec. 4 and the evaluation appears in Sec. 5. Related work is discussed in Sec. 6.

2 Model

We consider an asynchronous distributed system with \( n \) mutually trusting clients \( C_1, \ldots, C_n \) and a server \( S \). The communication between the server and the clients is reliable and respects first-in first-out (FIFO) semantics. Clients cannot communicate with each other. A protocol \( P \) specifies the behavior of the clients and the server. All clients are correct and hence follow \( P \); in particular, they do not crash (although crashes could be tolerated easily). The server is either correct and follows \( P \) or Byzantine, deviating arbitrarily from \( P \).

The clients invoke operations of a stateful functionality \( F \), implementing a set of deterministic operations; \( F \) defines a response and a state change for every operation. We use the standard notions of executions, histories, sequential histories, real-time order, concurrency, and well-formed executions from the literature [2]. In particular, every operation in an execution is represented by an invocation event and a response event. We extend \( F \) with a special return value \text{ABORT} that allows an operation to abort without taking effect, which may be used when concurrent operations would cause it to block [20, 8].

2.1 Consistency properties

When \( S \) is correct, the protocols should provide the standard notion of linearizability [17] with respect to \( F \). It requires that an execution of operations by all clients together is equivalent to an imaginary sequential execution of \( F \) that, furthermore, respects the real-time order among the operations.

Fork-linearizability [22, 9] relaxes this common global view and permits that the clients observe an execution that may split into multiple linearizable “forks,” which must never join again. More precisely, an execution is fork-linearizable when every client observes a linearizable history and for any operation observed by multiple clients, the history of events occurring before the operation is the same at all clients. This implies that if the views of the execution at two clients ever diverge, they cannot observe each other’s operations any more. This makes it easy to spot consistency violations by the server.

Furthermore, we recall the concept of a fork-linearizable Byzantine emulation [9]. It requires that our protocol among the clients and the Byzantine server satisfies two conditions: When the server is correct, then the service is linearizable; otherwise, it is still fork-linearizable. Finally, our protocol may only abort (by returning \text{ABORT}) if there is some reason for this; in other words, when the clients execute operations sequentially, then no client ever aborts.

2.2 Cryptographic primitives

Our protocols use cryptographic hash functions and digital signature schemes for protecting data against modification. We model them in an idealized way, as if implemented by a distributed oracle [6].

A cryptographic hash function \( \text{hash} \) maps a bit string \( x \) of arbitrary length to a short, unique hash value \( h \). Its implementation is deterministic and maintains a list \( L \) of all \( x \) that have been queried so
far. When the invocation contains \( x \in L \), then \( \text{hash} \) responds with the index of \( x \) in \( L \); otherwise, \( \text{hash} \) appends \( x \) to \( L \) and returns its index. This ideal implementation models only collision resistance, i.e., that it is not feasible to find two different values \( x_1 \) and \( x_2 \) such that \( \text{hash}(x_1) = \text{hash}(x_2) \). Real-world hash function implementations also satisfy one-wayness, that is, given hash value \( h \) it is infeasible to find an input \( x \) such that \( h = \text{hash}(x) \).

A digital signature scheme as used here provides two functions, \( \text{sign}_i \) and \( \text{verify}_i \), to ensure the authenticity of a message created by a known client. The scheme works as follows: A client \( C_i \) invokes \( \text{sign}_i(m) \) with a message \( m \) as argument and obtains a signature \( \phi \in \{0, 1\}^* \) with the response. Only client \( C_i \) can invoke \( \text{sign}_i \). When \( m \) and \( \phi \) are sent to another client, that client may verify the integrity of \( m \). It invokes \( \text{verify}_i(\phi, m) \) and obtains TRUE as a response if and only if \( C_i \) has executed \( \text{sign}_i(m) \); otherwise, \( \text{verify}_i(\phi, m) \) returns FALSE. Every client, as well as \( S \), may invoke \( \text{verify} \).

2.3 Authenticated data structures

This section defines the model of authenticated data structures (ADS) used here. Authenticated data structures [23, 21, 27] are a well-known tool for verifying operations and their results over data outsourced to untrusted servers. Popular instantiations rely on Merkle hash trees or other hierarchical authenticated search structures [16], for example.

We model an ADS for an arbitrary deterministic functionality \( F \). Departing from the literature on ADSs, we eliminate the special role of the single writer or “source” and let any client perform update operations; likewise, we unify queries and updates into one type of operation from a set \( O \). Operations may contain arguments according to \( F \), but these are subsumed into the different \( o \in O \). The functionality specifies a state \( s \in S \), which will be maintained by the server, starting with an initial state \( s_0 \). For example, this includes all data stored on a cloud storage service. Given \( s \), applying an operation \( o \) of \( F \) means to compute \( (s', r) \leftarrow F(s, o) \), resulting in a new state \( s' \in S \) and a response \( r \in R \).

Operations are executed by the clients and formally described by an invocation event and a response event occurring at the client. In order to verify the responses of \( S \), a client stores an authenticator \( a \), a short value also called a digest. Initially, the authenticator is a special value \( a_0 \).

For executing an operation \( o \), the client triggers algorithm \( \text{query}_F \) on \( S \),

\[
(r, \sigma_o) \leftarrow \text{query}_F(s, o),
\]

producing a response \( r \) and auxiliary data \( \sigma_o \), for \( o \); the latter may serve as a proof for the validity of the response. Then the client locally performs an operation \( \text{authexec}_F \) to obtain an output

\[
(a', \sigma'_o, v) \leftarrow \text{authexec}_F(o, a, r, \sigma_o).
\]

Here \( a' \) and \( \sigma'_o \) are the updated authenticator and auxiliary data, respectively, and \( v \in \{\text{FALSE, TRUE}\} \) denotes a Boolean verification value that tells the client whether the response \( r \) from \( S \) is valid.

The client should then send \( \sigma'_o \) back to \( S \), so that the server may actually execute \( o \) and update its state from \( s \) to \( s' \) by running \( \text{refresh}_F \) as

\[
s' \leftarrow \text{refresh}_F(s, o, \sigma'_o).
\]

Note we may also consider these operations for sequences of operations.

An ADS [27, 21, 24] is a special case of this formalization, in which the operations \( O \) can be partitioned into update-operations \( U \) and query-operations \( Q \). Update operations generate no response, i.e., \( F(s, u) = (s', \bot) \) for all \( u \in U \) and queries do not change the state, that is, \( F(s, q) = (s, r) \) for all \( q \in Q \).
Furthermore, an ADS may contain initialization and key-generation routines and all algorithms may take public and private keys as inputs in addition. For simplicity, and because our ADS implementations are unkeyed, we ignore them here.

An ADS satisfies correctness and security. Consider the execution of a sequence \( \langle o_1, \ldots, o_m \rangle \) of operations, \( F(s_0, \langle o_1, \ldots, o_m \rangle) \), which means to compute \( (s_j, r_j) \leftarrow F(s_{j-1}, o_j) \) for \( j = 1, \ldots, m \). A proper authenticated execution of \( \langle o_1, \ldots, o_m \rangle \) computes the steps

\[
(r_j, \sigma_{o,j}) \leftarrow \text{query}_F(s_{j-1}, o_j) \\
(a_j, \sigma'_{o,j}, v_j) \leftarrow \text{authexec}_F(o_j, a_{j-1}, r_j, \sigma_{o,j}) \\
s_j \leftarrow \text{refresh}_F(s_{j-1}, o_j, \sigma'_{o,j}),
\]

such that \( v_j = \text{TRUE} \), for \( j = 1, \ldots, m \).

An ADS is correct if the proper authenticated execution of any operation sequence \( \langle o_1, \ldots, o_m \rangle \) outputs state \( s_m \) and response \( r_m \) such that \( (s_m, r_m) = F(s_0, \langle o_1, \ldots, o_m \rangle) \).

Furthermore, an ADS must be secure against an adversary \( \mathcal{A} \) that tries to forge a response and auxiliary data that are considered valid by a client. More precisely, \( \mathcal{A} \) adaptively determines an operation sequence \( \langle o_1, \ldots, o_m \rangle \), which is taken through a proper authenticated execution by a challenger; at every step, \( \mathcal{A} \) obtains \( a_j \) and \( \sigma'_{o,j} \) and then determines \( o_{j+1} \) and whether the execution continues. Finally, after obtaining \( a_m \) and \( s_m, \mathcal{A} \) outputs an operation \( o^* \), a response \( r^* \), and a value \( \sigma^*_o \). The ADS is secure if no \( \mathcal{A} \) succeeds in creating \( o^*, r^*, s_m, \) and \( \sigma^*_o \) such that

\[
(\cdots, \text{TRUE}) = \text{authexec}_F(o^*, a_m, r^*, \sigma^*_o)
\]

but \( F(s_m, o^*) = (s_{m+1}, r_{m+1}) \) with \( r_{m+1} \neq r^* \). In other words, \( \mathcal{A} \) cannot find any \( o^* \) executed on \( s_m \) and forge a response and a proof for \( o^* \) that is accepted by the client, unless the response is the correct one.

Note that this formalization represents an “idealized” security notion. It is easy to formulate an equivalent computational security condition using the language of modern cryptology [14]. This model subsumes the one of Cachin [5] and generalizes ADS [27] to multiple writing clients.

### 2.4 Compatible operations

Our protocol takes advantage of compatible operations that permit “concurrent” execution without compromising the goal of ensuring fork-linearizability to the clients. An operation \( o' \) is compatible with another operation \( o \) (in a state \( s \)) if the presence of \( o' \) before \( o \) does not influence the return value of \( o \) (in \( s \)). Compatible operations can be executed without blocking; this improves the throughput compared to earlier protocols, in particular with respect to COP [8], which considered the stronger notion of commutative operations.

Formally, we say an operation sequence \( \omega \) and an operation \( o \) are compatible in a state \( s \) whenever the responses of \( o \) remain the same regardless of whether \( \omega \) executed before \( o \). Hence, with

\[
(s', r) \leftarrow F(s, \omega); \quad (s'', p) \leftarrow F(s', o); \quad \text{and} \quad (t', q) \leftarrow F(s, o)
\]

it holds \( p = q \). Moreover, we say that \( \omega \) is compatible with \( o \) if and only if \( \omega \) and \( o \) are compatible in all states \( s \in \mathcal{S} \) of \( F \). Note that compatible operations may not commute.

In algorithms we use a function \( \text{compatible}_F \), which takes \( \omega \) and \( o \) as inputs and returns \text{TRUE} if and only if \( \omega \) and \( o \) are compatible. We can also extend this notion to sequences of operations. Two sequences \( \rho_1 \) and \( \rho_2 \) consisting of operations in \( \mathcal{O} \) are compatible if and only if \( \rho_1 \) is compatible with every single operation in \( \rho_2 \).
3 The ADS integrity protocol

This section introduces the ADS integrity protocol (AIP), a generic protocol to verify the integrity and consistency for any authenticated data structure (ADS) AIP extends and improves upon ACOP proposed by Cachin and Ohrimenko [8]. Sec. 4 shows how to instantiate AIP for a cloud storage application, and the evaluation in Sec. 5 demonstrates the advantage of AIP over ACOP.

3.1 Overview

The processing of one operation in AIP is structured into an active and a passive phase, as shown in Fig. 1. The active phase begins when the client invokes an operation and ends when the client completes it and outputs a response; this takes one message roundtrip between the client and the server. The client stays further involved with processing authentication data for this operation during the passive phase, which is decoupled from the execution of further operations.

More precisely, when client $C_i$ invokes an operation $o \in O$, it sends a signed INVOKE message carrying $o$ to the server $S$. The server assigns a global sequence number ($t$) to $o$ and responds with a REPLY message containing a list of pending operations, the response, an authenticator, and auxiliary data needed by the client for verification. Operations are pending (for $o$) because they have been started by other clients and $S$ has ordered them before $o$, but $S$ has not yet finished processing them. We distinguish between pending-other operations, which have been invoked by other clients, and pending-self operations, which $C_i$ has executed before $o$.

After receiving the REPLY message, the client checks its content. In particular, if the pending-other operations are compatible with $o$, then $C_i$ verifies the pending-self operations including $o$ with the help of the authenticator; if they are correct, $C_i$ outputs the response. Along the way $C_i$ verifies that all data received from $S$ satisfies conditions to ensure fork-linearizability. An operation that terminates like this is called successful; alternatively, when the pending-other operations are not compatible with $o$, then $o$ aborts. In this case, $C_i$ returns the symbol ABORT. In any case, the client subsequently commits $o$ and sends a signed COMMIT message to $S$. At this point, the client may invoke the next operation or retry $o$.

Processing of $o$ continues with the passive phase. Later, when the operation immediately preceding $o$ in the assigned order has terminated its passive phase, $S$ sends an UPDATE-AUTH message with auxiliary data and the authenticator of the preceding operation to $C_i$. When $C_i$ receives this, it validates the message content and verifies the execution of $o$ unless $o$ had been aborted. Using the methods of the ADS, the client now computes and signs a new authenticator that it sends to $S$ in a COMMIT-AUTH message. We say that $C_i$ authenticates $o$ at this time. When $S$ receives this message, $S$ applies $o$ by executing it on the state and stores the corresponding authenticator; this completes the passive phase of $o$. 
Note that the server may receive COMMIT messages in an order that differs from the one of the globally assigned sequence numbers due to asynchrony. Still, the authentication steps in the passive phases of the different operations must proceed according to the assigned operation order. For this reason, the server maintains second sequence number \( b \), which indicates the last authenticated operation that the server has applied to its state. Hence, \( S \) buffers the incoming COMMIT messages and runs the passive phases sequentially in the assigned order.

For ensuring consistency, every client needs to know about all operations that the server has executed. Therefore, when \( S \) responds to the invocation of an operation by \( C_i \), it includes in the REPLY message a summary (including the corresponding signatures) of all those operations that \( C_i \) has missed since it last executed an operation. Prior to committing \( o \), the client verifies these operations and thereby clears them.

### 3.2 Notation

The protocol is shown in Alg. 1–3 and formulated reactively. The clients and the server are state machines whose actions are triggered by events such as receiving messages. An ordered list with elements \( e_1, e_2, \ldots, e_k \) is denoted by \( E = \langle e_1, e_2, \ldots, e_k \rangle \); the element with index \( j \) may be accessed as \( E[j] \).

We also use maps that operate as associative arrays and store values under unique keys. A value \( v \) is stored in a map \( H \) by assigning it to a key \( k \), denoted by \( H[k] \leftarrow v \); for non-assigned keys, the map returns \( \bot \). When only a subset of the elements in a tuple are needed, a “don’t-care” variable is denoted by “\( \cdot \)” and the symbol \( \| \) denotes the concatenation of bit strings. The assert statement, parameterized by a condition, signals an error that immediately terminates the protocol when the condition is false. Clients use this to signal that the server misbehaved.

### 3.3 Data structures

This section describes the state maintained by every client and by the server. For simplicity, the pseudo code does not describe garbage collection, but we note where this is possible.

#### 3.3.1 Client

Every client \( C_i \) stores the sequence number of its last cleared operation in a variable \( c \). The hash chain \( H \) represents the condensed view that \( C_i \) has of the sequence of all operations. It is computed over the sequence of all applied operations and the sequence of pending operations announced by \( S \). Formally, \( H \) is a map indexed by operation sequence number; an entry \( H[l] \) is equal to \( \text{hash}(H[l - 1]\|o\|l\|j) \) when the \( l \)-th operation \( o \) is executed by \( C_j \), with \( H[0] = \text{NULL} \). Variable \( Z \) is a map that represents the status (SUCCESS or ABORT) of every operation, according to the result of the test for compatibility. The client only needs the entries in \( H \) and \( Z \) with indices greater than \( c \) and may garbage-collect older entries. Finally, \( C_i \) uses a variable \( u \) that is set to \( o \) whenever \( C_i \) has invoked operation \( o \) but not yet completed it; otherwise \( u \) is \( \bot \).

#### 3.3.2 Server

The server maintains the sequence number of the most recently invoked operation in a counter \( t \). In addition to that, the counter \( b \) contains the sequence number of the most recently applied operation and governs the authentication of operations in the passive phase. Every invoked operation is stored in a map \( I \) and every committed operation in a map \( O \); both maps are indexed by sequence number. The server only needs the entries in \( I \) with sequence numbers greater than \( b \). An entry in \( O \), at a sequence number \( b \) or greater, has to be stored until every client has committed an operation with a higher sequence number.
number and may be removed later. Most importantly, the server keeps the state $s$ of the ADS for $F$, which reflects all successful operations up to sequence number $b$. Note that in previous work [8] every client maintains a complete copy of the state $S$. Moreover, $S$ stores the authenticator for every operation in a map $A$ indexed by sequence number.

### 3.4 The protocol in detail

This section describes the ADS integrity protocol (AIP) as shown in Alg. 1–3. AIP is parameterized by an ADS and a functionality $F$ that specifies its operations through $query_F$, $authexec_F$, and $refresh_F$. The client invokes AIP with an ADS-operation $o$ by calling $aip-invoke(o)$; it completes when AIP executes $\text{return}$ at the end of the handler for the $\text{REPLY}$ message. This ends the active phase of AIP, and the passive phase continues asynchronously in the background.

#### 3.4.1 Active phase

When client $C_i$ invokes an operation $o$, it computes an $\text{INVOKE}$-signature $\tau$ over $o$ and $i$; this proves to other clients that $C_i$ has invoked $o$. Then $C_i$ stores $o$ in $u$ and sends an $\text{INVOKE}$ message with $o$ and $\tau$ to the server.

Upon receiving an $\text{INVOKE}$ message with $o$, the server increments the sequence number $t$, assigns it to $o$, and assembles the $\text{REPLY}$ message for $C_i$. First, $S$ stores $o$ and the accompanying signature in $I[t]$. The pending operations for $o$, assigned to $\omega$, are found in $I[b+1], \ldots, I[t]$, i.e., starting with the oldest non-authenticated operation, and include $o$. In order to compute the response and the auxiliary data for $o$ from the correct state, the server must then extract the successful pending-self operations $\mu$ of $C_i$, using the following method:

```plaintext
function separate-pending(i, $\omega$)
    $\mu \leftarrow \langle \rangle$; $\gamma \leftarrow \langle \rangle$
    for $k = 1, \ldots, \text{length}(\omega)$ do
        $(o', \cdot, \cdot) \leftarrow \omega[k]$
        if $j = i$ then
            if $k = \text{length}(\omega)$ $\land$ status of $o'$ is SUCCESS then
                $\mu \leftarrow \mu \circ \langle o' \rangle$
            else if $j \neq i$ then
                $\gamma \leftarrow \gamma \circ \langle o' \rangle$
        return $(\mu, \gamma)$
```

This method is common to the server and the clients. Note that $\mu$ includes the current operation (which appears at the end of $\omega$) but not the aborted operations of $C_i$. The server finds the status of a pending-self operation $o'$ of $C_i$ in $O[b+k]$ (except for $o$ itself, obviously) because $C_i$ has already committed $o'$ prior to invoking $o$ and because the messages between $C_i$ and $S$ are FIFO-ordered. On the other hand, $C_i$ retrieves the status of $o'$ from $Z[b+k]$.

Then $S$ computes the response $r$ and auxiliary data $\sigma_o$ by calling $query_F(s, \mu)$ from the ADS for $F$; the response therefore takes into account the state reached after the successful pending-self operations of $C_i$ but excludes any pending-other operations present in $\omega$. However, the client will only execute $o$ and output $r$ when $\mu$ is compatible with $o$ and, therefore, $C_i$ is guaranteed a view in which the operations of $\mu$ occur after $o$. This will ensure fork-linearizability. The $\text{REPLY}$ message to $C_i$ also includes $A[b]$ containing the authenticator and its $\text{AUTH}$-signature, for the operation at position $b$. The client passes these to $authexec_F$ of $\mu$ for verifying the correctness of the response. Furthermore, the $\text{REPLY}$ message contains $\delta$, the list of all operations that have been authenticated since $C_i$’s last operation. In particular,
Algorithm 1 ADS integrity protocol (AIP) for client C_i

state
\( c \in \mathbb{N}_0 \): sequence number of last cleared operation, initially 0
\( H : \mathbb{N}_0 \rightarrow \{0, 1\}^* \): hash chain, initially only \( H[0] = \perp \)
\( Z : \mathbb{N}_0 \rightarrow Z \): status map, initially empty
\( u \in \mathcal{O} \cup \{\perp\} \): current operation or \( \perp \) if none, initially \( \perp \)

function aip-invoke(o)
\( u \leftarrow o \)
\( \tau \leftarrow \text{sign}_i(\text{INVOKE}\|o\|i) \)
send message \([\text{INVOKE}, o, \tau, c]\) to S

upon receiving message \([\text{REPLY}, \delta, b, \alpha, t, r, \sigma_o]\) from S do
\( \text{check-view}(\delta, b, \alpha) \)
\( \text{check-pending}(\omega) \)
\( (\mu, \gamma) \leftarrow \text{separate-pending}(i, \omega) \)
\( t \leftarrow b + \text{length}(\omega) \)
if compatible_F(\gamma, u) then
\( (\cdot, \cdot, v) \leftarrow \text{authexec}_F(\mu, a, r, \sigma_o) \)
assert v
\( Z[t] \leftarrow \text{SUCCESS} \)
else
\( r \leftarrow \text{ABORT} \)
\( Z[t] \leftarrow \text{ABORT} \)
\( \phi \leftarrow \text{sign}_i(\text{COMMIT}\|t\|u\|i\|Z[t]\|H[t]) \)
send message \([\text{COMMIT}, u, t, Z[t], \phi]\) to S
\( u \leftarrow \perp \)
return \( r \) // response of operation aip-invoke(o)

upon recv. msg. \([\text{UPDATE-AUTH}, o, r, \sigma_o, \phi, q, \delta, \alpha]\) from S do
assert verify_j(\phi, \text{COMMIT}\|q\|o\|i\|Z[q]\|H[q])
\( (\alpha_h, \cdot, \cdot, j) \leftarrow \delta \)
\( (a, \psi) \leftarrow \alpha \)
assert verify_j(\psi, \text{AUTH}\|\alpha_h\|q-1\|H[q-1]\|a)
if Z[q] = \text{SUCCESS} then
\( (a', \sigma'_o, v) \leftarrow \text{authexec}_F(o, a, r, \sigma_o) \)
assert v
else
\( (a', \sigma'_o) \leftarrow (a, \perp) \)
\( \psi' \leftarrow \text{sign}_j(\text{AUTH}\|o\|q\|H[q]\|a') \)
send message \([\text{COMMIT-AUTH}, a', \sigma'_o, \psi']\) to S
Algorithm 2 ADS integrity protocol (AIP) for client $C_1$, continued

function extend-chain$(o, l, j)$
if $H[l] = \perp$ then
\[ H[l] \leftarrow \text{hash}(H[l-1]||o||l||j) \] // extend by one
else if $H[l] \neq \text{hash}(H[l-1]||o||l||j)$ then
return FALSE // server replies are inconsistent
return TRUE

function check-view$(\delta, b, \alpha)$
assert $\text{length}(\delta) \geq 1$
if $b = c$ then $d \leftarrow c - 1$ else $d \leftarrow c$
for $k = 1, \ldots, \text{length}(\delta)$ do
\[ l \leftarrow d + k \]
\[ (o, z, \phi, j) \leftarrow \delta[k] \]
assert extend-chain$(o, l, j)$
assert verify$_j(\phi, \text{COMMIT}||o||j||z||H[l])$
\[ (a, \psi) \leftarrow \alpha \] // variables $o$ and $j$ keep their values
assert verify$_j(\psi, \text{AUTH}||o||b||H[b]||a)$
\[ c \leftarrow d + \text{length}(\delta) \] // all operations in $\delta$ have been cleared

function check-pending$(\omega)$
assert $\text{length}(\omega) \geq 1$
for $k = 1, \ldots, \text{length}(\omega)$ do
\[ l \leftarrow c + k \]
\[ (o, \tau, j) \leftarrow \omega[k] \]
assert extend-chain$(o, l, j) \land \text{verify}_j(\tau, \text{INVOKE}||o||j)$
assert $o = u \land j = i$ // variables $o$ and $j$ keep their values

when $c$ is the sequence number from the INVOKE message, $\delta$ contains the operations at positions $c + 1, \ldots, b$; when $c = b$, however, $\delta$ contains still $O[b]$.

After receiving the REPLY message from $S$, the client (1) processes and clears the authenticated operations in $\delta$, (2) verifies the pending operations in $\omega$, $\mu$, and $\gamma$, and (3) verifies that $r$ is the correct response for $o$.

For verifying and processing $\delta$ and the last signed authenticator in $\alpha$, $C_i$ calls a function check-view. It verifies and/or extends the hash chain for every operation and verifies the corresponding COMMIT-signature. Finally, it also checks the AUTH-signature on the authenticator $a$, which is contained in $\alpha$. If successful, all operations in $\delta$ are cleared and $C_i$’s operation counter $c$ is advanced to the position of the last operation in $\delta$. (The check for $b = c$ ensures that $\delta$ contains at least one operation at position $c$.)

The client continues in check-pending by verifying that the pending operations are announced correctly: for every operation in $\omega$, it determines the sequence number $l$, verifies the corresponding INVOKE-signature $\tau$, and checks the hash chain entry $H[l]$. If there is no entry in $H$ for $l$, then $C_i$ computes the new entry from $o$, $l$, and $H[l - 1]$; otherwise, $C_i$ verifies the that existing entry matches the expected value. If this validation succeeds, it means the operation is consistent with a pending operation sent previously by $S$. After iterating through the pending operations, the client checks also that the last operation in $\omega$ is indeed its own current operation $o$.

Next, $C_i$ invokes separate-pending to extract $\mu$ and $\gamma$ from $\omega$ (see earlier). Then, $C_i$ checks whether $\gamma$ is compatible with $u$ (the last invoked operation). If yes, $C_i$ calls the ADS operation authexec$_F((\mu, a, r, \sigma_o))$ for verifying that applying the operations in $\mu$ yields $r$ as the response (recall that $\mu$ includes $o$ at the end). This is only a simulated execution, as the authenticator and auxiliary data output by authexec$_F$ are
Algorithm 3 ADS integrity protocol (AIP) for server $S$

Let

- $t \in \mathbb{N}_0$: seqno. of last invoked op., initially 0
- $b \in \mathbb{N}_0$: seqno. of last applied op., initially 0
- $I : \mathbb{N} \to \mathcal{O} \times \{0, 1\}^* \times \mathbb{N}$: invoked ops., initially empty
- $O : \mathbb{N} \to \mathcal{O} \times \{0, 1\}^* \times \mathbb{Z} \times \{0, 1\}^* \times \mathbb{N}$: committed ops., initially empty
- $A : \mathbb{N}_0 \to \{0, 1\}^*$: authenticators, init. $A[0] = a_0$
- $s \in \{0, 1\}$: state of the service, initially $s = s_0$

Upon receiving message $[\text{INVOKE}, o, \tau, c]$ from $C_i$

- $t \leftarrow t + 1$
- $I[t] \leftarrow (o, \tau, i)$
  
  **if** $b = c$ **then** $\delta \leftarrow \langle O[b] \rangle$ **else** $\delta \leftarrow \langle O[c + 1], \ldots, O[b] \rangle$
  
  $\omega \leftarrow \langle I[b + 1], I[b + 2], \ldots, I[t] \rangle$

  // all pending operations

  $(\mu, \cdot) \leftarrow \text{separate-pending}(i, \omega)$
  
  $(r, \sigma_o) \leftarrow \text{query}_F(s, \mu)$

  send message $[\text{REPLY}, \delta, b, A[b]]$ to $C_i$

Upon receiving message $[\text{COMMIT}, o, q, z, \phi]$ from $C_i$

- $O[q] \leftarrow (o, z, \phi, t)$

Upon $O[b + 1] \neq \bot \land A[b + 1] = \bot$ do

  $(o, z, \phi, j) \leftarrow O[b + 1]$

  **if** $z = \text{SUCCESS}$ **then**
  
  $(r, \sigma_o) \leftarrow \text{query}_F(s, o)$

  **else**
  
  $(r, \sigma_o) \leftarrow (\bot, \bot)$

  send msg. $[\text{UPDATE-AUTH}, o, r, \sigma_o, \phi, b + 1, O[b], A[b]]$ to $C_j$

Upon receiving message $[\text{COMMIT-AUTH}, a, \sigma_o, \psi]$ from $C_i$

- $b \leftarrow b + 1$
- $A[b] \leftarrow (a, \psi)$

  $(o, z, \cdot, \cdot) \leftarrow O[b]$

  **if** $z = \text{SUCCESS}$ **then**
  
  $s \leftarrow \text{refresh}_F(s, o, \sigma_o)$

ignored. Finally, $C_i$ commits $o$ by generating a COMMIT-signature over $t$, the sequence number of $o$, its status, and its hash chain entry, sends a COMMIT message (with $t$, the operation, and the signature) to $S$, and outputs the response $r$.

### 3.4.2 Passive phase

The server buffers the content of all incoming COMMIT messages in $O$ and processes them in the order of their sequence number, indicated by $b$. When an operation with sequence number $b + 1$ has been committed (i.e., upon $O[b + 1] \neq \bot \land A[b + 1] = \bot$), the server uses $\text{query}_F$ to compute the response $r$ and to extract the auxiliary data $\sigma_o$ from the current state $s$. It sends this in an UPDATE-AUTH message to $C_i$, also including the operation at position $b$ (from $O[b]$) and its authenticator (taken from $A[b]$), which have been computed before. These values allow the client to verify the authenticity of the response for the operation at position $b + 1$.

When processing an UPDATE-AUTH message for an operation $o$ and sequence number $q$, the client
first validates the message contents. In particular, \( C_i \) verifies that the authenticator \( a \) is covered by a valid \textit{AUTH}-signature by client \( C_j \) with sequence number \( q - 1 \), using \( C_i \)'s hash chain entry \( H[q - 1] \).

Next, if \( o \) was not aborted, i.e., \( Z[q] = \text{SUCCESS} \), the client invokes \textit{authexec} to verify that the auxiliary data and the response are correct, and to generate new auxiliary data \( s'_o \) and a new authenticator \( a' \), which vouches for the correctness of the state updates induced by \( o \). Otherwise, \( C_i \) skips this step, as the authenticator does not change. Then \( C_i \) issues an \textit{AUTH}-signature \( \psi' \) and sends it back to \( S \) together with \( a' \) and \( s'_o \) in a \textit{COMMIT-AUTH} message.

As the last step in the passive phase, \( S \) increments \( b \), stores the data in the \textit{COMMIT-AUTH} message at \( A[b] \), and if the operation did not abort, \( S \) applies it to \( s \) through \textit{refreshF}.

### 3.5 Remarks

As in BST [28] and in COP [8], operations that do not interfere with each other may proceed without blocking. Non-compatible operations are aborted and must be retried later. Such wait-free semantics are highly desirable but cannot always be guaranteed [9]. The potential for blocking has led other systems, including SPORC [12] and FAUST [7], to adopt weaker and less desirable guarantees than fork-linearizability.

Obviously, it makes no sense for a client to retry its operation while the non-compatible operation is still pending. However, the client does not know when the contending operation commits. Additional communication between the server and the clients could be introduced to signal this. Alternatively, the client may employ a probabilistic waiting strategy and retry after a random delay.

In the following we assume that \( S \) is correct. The communication cost of AIP amounts to the five messages per operation. Every client eventually learns about all operations of all clients, as it must clear them and include them in its hash chain. However, this occurs only when the client executes an operation (in \textit{REPLY}). At all other times between operations, the client may be offline and inactive. In a system with \( n \) clients that performs \( h \) operations in total, BST [28] and COP [8] require \( \Theta(nh) \) messages overall. AIP reduces this cost to \( \Theta(h) \) messages, which means that each client only processes a small constant number of messages per operation.

The size of the \textit{INVOKE}, \textit{COMMIT}, \textit{UPDATE-AUTH}, and \textit{COMMIT-AUTH} messages does not depend on the number of clients and on the number operations they execute. The size of the \textit{REPLY} message is influenced by the amount of contention, as it contains the pending operations. If one client is slow, the pending operations may grow with the number of further operations executed by other clients. Note that the oldest pending operation is the one at sequence number \( b + 1 \); hence, all operations ordered afterwards are treated as pending, even when they already have been committed. The \textit{REPLY} message can easily be compressed to constant size, however, by omitting the pending operations that have already been sent in a previous message to the same client.

The functionality-dependent cost, in terms of communicated state and auxiliary data, is directly related to the ADS for \( F \). In practice, hierarchical authenticated search structures, such as hash trees and skip lists, permit small authenticators and auxiliary data and are very efficient [11].

### 3.6 Correctness

We consider three cases: (1) \( S \) is correct and the clients execute operations (1a) sequentially or (1b) concurrently; and (2) \( S \) is malicious.

In case (1a), all operations execute one after each other. When, furthermore, the \textit{COMMIT-AUTH} message from a client reaches \( S \) before the next operation is invoked, then AIP is similar to “serialized” SUND\textsc{r} [19] and the “lock-step protocol” of Cachin [5]. This means that a client \( C_i \) executing an operation \( o \) receives a \textit{REPLY} message with all authenticated operations that exist in the system and
only \( o \) as pending operation. Then \( t = b + 1 \), and later \( C_i \) commits \( o \) and authenticates \( o \) without further operations intermixed at \( S \) nor at any client. Clearly, this execution is linearizable and satisfies the first condition of a fork-linearizable Byzantine emulation.

In case (1b), there may exist pending-other operations, but since \( C_i \) verifies that its own operations are compatible with them, the response value is correct. As \( S \) is correct, the views of all clients are equal, i.e., prefixes of each other, and this ensures linearizability.

For case (2), note that every client \( C_i \) starts to extend its view from a cleared, authenticated operation and the corresponding signed authenticator \( a \). If the pending-other operations in \( \gamma \) commute with \( o \) and the response is valid w.r.t. \( a \) (according to \( \text{authexec}_F \)), then it is safe for \( C_i \) to output the response and thereby include it into its view. The malicious \( S \) may order the operations in \( \gamma \) differently at other clients, creating a fork, but they can be omitted from the view of \( C_i \). The hash chain maintained by \( C_i \) contains a condensed representation of its entire view. By using its own hash chain to verify all COMMIT and AUTH signatures of other clients, \( C_i \) ensures that the views of these other clients are equal, hence, whenever an operation \( o \) appears in the views of two clients, also their views are the same up to \( o \). This ensures fork-linearizability.

### 3.7 Extensions

In order to keep the complexity of the protocol description for AIP on a comprehensible level, we present two important efficiency improvements here as informal extensions.

#### 3.7.1 Batching AUTH-UPDATE messages

Recall that the clients authenticate operations in the order of their sequence numbers. Some client \( C_{\text{slow}} \) may fall behind with this step, and when the other clients proceed faster and execute more operations, the number of pending operations grows continuously. This creates much more work for the faster clients for processing the REPLY message and slows them down.

However, since all clients trust each other, one can modify AIP such that another client \( C_{\text{fast}} \) may step in for \( C_{\text{slow}} \), handle the UPDATE-AUTH message, and sign the authenticator for the operation of \( C_{\text{slow}} \). Only small changes to the data structures are needed to accommodate this change. Ideally \( C_{\text{fast}} \) has more processing power or is closer to \( S \) on the network than \( C_{\text{slow}} \); this choice should be determined heuristically based on the measured performance or network delays.

Extending the above idea, \( S \) might actually batch all non-authenticated operations when an operation from \( C_i \) commits at sequence number \( q \). Hence, \( S \) sends the UPDATE-AUTH messages for all operations between \( b \) and \( q \) to \( C_i \) and has \( C_i \) authenticate them. This works because a client can authenticate two consecutive operations without going back to \( S \). The server may then record the COMMIT-AUTH responses from the fastest client and inform the others that their help with authenticating operations is no longer needed.

#### 3.7.2 Eliminating aborted operations

A second extension removes aborted operations from the pending operations. Recall that in AIP, an operation \( o \) resides in the list of pending operations until the client has committed the authenticator of \( o \), even if the operation was aborted. This may have the negative effect that later operations may not be compatible with \( o \) and abort. However, if \( o \) has been aborted and committed by the client, and \( S \) has received its COMMIT message, then \( S \) can include this with the list of pending operations sent for a later operation \( o' \). The client that executes \( o' \) will take into account that \( o \) was aborted and ignore it for determining whether \( o' \) is compatible with the pending operations. Depending on the workload, this reduces further aborts.
4 Verification of integrity and consistency of cloud object storage (VI-COS)

This section introduces our main contribution, the protocol for verifying the integrity and consistency of cloud object storage, abbreviated VICOS. It leverages AIP from the previous section and provides a fork-linearizable Byzantine emulation for a practical object-store service, in a manner that is transparent to the storage provider. We first define the operations of the cloud storage service and outline the architecture of VICOS. Next we instantiate AIP for verifying the integrity of a simple object store and show how VICOS extends this to practical cloud storage.

More precisely, VICOS consists of the following components (see Fig. 2):

1. A cloud object store service, as offered by commercial providers. It maintains the object data stored by the clients using VICOS.

2. An AIP client and an AIP server, which implement the protocol from the previous section for the functionality of an authenticated dictionary (AD) and authenticate the objects at the cloud object store. The AIP server runs remotely as a cloud service accessed by the AIP client. We also refer to this as AD-AIP.

3. The VICOS client exposes a cloud object store interface to the client application and performs integrity and consistency verification. During each operation, the client consults the cloud object store for the object data itself and the AIP server for integrity-specific metadata. In particular, the AD-AIP stores a cryptographic digest of every object.

Note that the cloud object store as well as the AIP server are in the untrusted domain; they may, in fact, collude together against the clients.

![Figure 2. Architecture of VICOS: the two components of the cloud service are shown at the top, the client at the bottom.](image)

4.1 Object store

An object store or key-value store (KVS) provides a “simple” storage service to multiple clients. It stores a practically unbounded number of objects in a flat namespace, where each object is an arbitrary sequence of bytes (or a “blob,” a binary large object), identified by a unique name or key. We assume that clients may only read and write entire objects, i.e., it is not possible to read from or write into the middle of an object, as in a file system.

Our formal notion of a KVS internally maintains a map $M$ that stores the values in $V$ under their respective keys taken from a universe $K$. It provides four operations:
1. \texttt{kvs-put}(k, v): Stores a value \(v \in \mathcal{V}\) under key \(k \in \mathcal{K}\), that is, \(M[k] \leftarrow v\).

2. \texttt{kvs-get}(k): Returns the value stored under key \(k \in \mathcal{K}\), that is, \(M[k]\).

3. \texttt{kvs-del}(k): Deletes the value stored under key \(k \in \mathcal{K}\), that is, \(M[k] \leftarrow \perp\).

4. \texttt{kvs-list}(): Returns a list of all keys that are associated with a value, that is, the list \(\langle k \in \mathcal{K} | M[k] \neq \perp \rangle\).

This KVS interface forms the core of many real-world cloud storage services, such as Amazon S3 or OpenStack Swift. Typically there is a bound on the length of the keys, such as a few hundred bytes, but the stored values can be much larger (e.g., many Gigabytes). For simplicity, we assume that the object store provides atomic semantics during concurrent access, being aware that cloud storage systems may only be eventually consistent [3] due to network partitions.

Many practical cloud object stores support a single-level hierarchical name space, formed by containers or buckets. We abstract this separation into the keys here; however, a production-level system would introduce this separation again by applying the design of VICOS for every container.

### 4.2 Authenticated dictionary integrity protocol (AD-AIP)

VICOS instantiates AIP with the functionality of a KVS that stores only short values. In order to distinguish this function from the cloud object store, we refer to it as authenticated dictionary (AD) [23], denote it by AD-AIP, and use \texttt{ad-put}, \texttt{ad-get}, etc. for its operations.

The implementation of AD-AIP uses the well-known approach of building hash tree over its entries [4, 23, 11]; see Alg. 4–5 for the details of the AIP instantiation to AD. The AD-AIP server stores the values in a map \(D\) and maintains a hash tree \(HT\), constructed over the list of key-value pairs stored in the map, according to a fixed sort order on the keys. That is, every leaf node of the hash tree is computed by hashing the node key, its value, and the key of the successor leaf node together. The next node has to be included in order to authenticate the absence of a key in response to a query for a non-existing key (e.g., [23]). The root of the hash tree serves as the authenticator for AD-AIP.

For the \texttt{ad-get} and \texttt{ad-list} operations, the server runs \texttt{query}_{AD-AIP} and extracts those paths from \(HT\) that are necessary to verify the correctness of the retrieved value and places them in \(s_o\). For \texttt{ad-put} and \texttt{ad-del} operations, \texttt{query}_{AD-AIP} also places these paths into \(s_o\) because the client needs them to construct the updated root hash. The asterisks (*) in Alg. 4–5 denote the additional data and steps necessary to verify the predecessor or successor leaves for authenticating an absent key (details of this are in the literature [11]).

The compatible_{AD-AIP}(\mu, u) function of Alg. 4–5 defines the compatibility relation among the operations of the authenticated dictionary; VICOS supports the same KVS interface and inherits this notion of compatibility. For more general services like databases, one would invoke a transaction manager here. For AD-AIP, the compatibility between an already executed operation \(u\) and a candidate operation \(o\) is given by \(T_{AD}(u, o)\) in Table 1. For instance, a \texttt{ad-put} followed by a \texttt{ad-get} for the same key or followed by \texttt{ad-list} is not compatible, whereas \texttt{ad-list} and \texttt{ad-get} operations are always compatible.

The advantage of VICOS over ACOP [8] becomes apparent here: only 8 pairs among the 49 cases shown are not compatible, whereas in ACOP, 22 among 49 cases do not commute and would be aborted.

### 4.3 Implementation

VICOS emulates the key-value store API of a cloud object store (COS) to the client and transparently adds integrity and consistency verification. As with AIP, consistency or data integrity violations committed by the server are detected through \texttt{assert}; any failing assertion triggers an alarm. It must be
Algorithm 4 Authenticated dictionary (AD-AIP), Part 1

state
\[ D : K \rightarrow \{0, 1\}^* : \text{authenticated dictionary, initially empty} \]
\[ HT : \text{hash tree over } d, \text{ initially empty} \]

function \text{query}_{AD-AIP}((D, HT), o)
\begin{align*}
\text{if } o &= \text{ad-put}(k, v) \lor o = \text{ad-del}(k) \text{ then} \\
& r \leftarrow \perp \\
& s_o \leftarrow \text{sibling nodes on path (*) from } k \text{ to root in } HT \\
\text{else if } o &= \text{ad-get}(k) \text{ then} \\
& r \leftarrow D[k] \\
& s_o \leftarrow \text{sibling nodes on path (*) from } k \text{ to root in } HT \\
\text{else if } o &= \text{ad-list()} \\
& r \leftarrow \langle k \in K | D[k] \neq \perp \rangle \\
& s_o \leftarrow HT \\
\text{return } (r, s_o)
\end{align*}

function \text{authexec}_{AD-AIP}(o, a, r, s_o)
\begin{align*}
\text{if } o &= \text{ad-put}(k, v) \lor o = \text{ad-get}(k) \lor o = \text{ad-del}(k) \text{ then} \\
& \text{if } s_o \text{ is not a valid path (*) from } k \text{ to tree root } a \text{ then} \\
& \text{return } (\cdot, \cdot, \text{FALSE}) \\
\text{if } o &= \text{ad-put}(k, v) \text{ then} \\
& \text{insert leaf node } k \text{ with value } v \text{ in the tree} \\
& s_o' \leftarrow \text{updated path from } k \text{ to tree root} \\
& a' \leftarrow \text{updated hash-tree root} \\
\text{else if } o &= \text{ad-get}(k) \text{ then} \\
& \text{if path (*) not consistent with node } k \text{ holding } r \text{ then} \\
& \text{return } (\cdot, \cdot, \text{FALSE}) \\
& s_o' \leftarrow \perp \\
& a' \leftarrow a \\
\text{else if } o &= \text{ad-del}(k) \text{ then} \\
& \text{delete leaf node } k \text{ from the tree} \\
& s_o' \leftarrow \text{updated paths from siblings of } k \text{ to tree root} \\
& a' \leftarrow \text{updated hash-tree root} \\
\text{else if } o &= \text{ad-list()} \\
& \text{if } r \text{ is not list of keys in leaves of tree with root } a \text{ then} \\
& \text{return } (\cdot, \cdot, \text{FALSE}) \\
& s_o' \leftarrow \perp \\
& a' \leftarrow a \\
\text{return } (a', s_o', \text{TRUE})
\end{align*}

followed by a recovery action whose details go beyond the scope of this paper. Analogously to AIP, VICOS may return \text{ABORT}; this means that the operation was not executed and the client should retry it.

Algorithm 6 presents the pseudo code of VICOS. Basically, it protects every object in the COS by storing its hash in the authenticated directory (AD). Operations on the object store trigger corresponding operations on COS and on AD, and AD-AIP is responsible for consistency enforcement.

In order to prevent race conditions, VICOS does not store the hash of an object under the object’s key in COS directly, but translates every object key to a unique key for COS. Otherwise, two concurrent operations accessing the same object might interfere with each other and leave the system in an inconsistent state. More precisely, in a \text{vicos-put}(k, v) operation, the client chooses a nonce \( x \) (a value guaranteed
Algorithm 5 Authenticated dictionary (AD-AIP), Part 2

function $\text{refresh}_{\text{AD-AIP}}((D, HT), o, s_o)$
  \begin{algorithmic}
  \STATE \textbf{if} $o = \text{ad-put}(k, v)$ \textbf{then}
  \STATE $D[k] \leftarrow v$
  \STATE update path in $HT$ from $k$ to root, as taken from $s_o$
  \STATE \textbf{else if} $o = \text{ad-del}(k)$ \textbf{then}
  \STATE $D[k] \leftarrow \perp$
  \STATE update path in $HT$ from $k$ to root, as taken from $s_o$
  \RETURN $(D, HT)$
  \end{algorithmic}

function $\text{compatible}_{\text{AD-AIP}}(u, \mu)$
  \begin{algorithmic}
  \FOR{$o \in \mu$}
  \STATE \textbf{if} $\neg T_{\text{AD}}(u, o)$ \textbf{then}
  \STATE \RETURN FALSE
  \STATE \RETURN TRUE
  \ENDFOR
  \RETURN TRUE
  \end{algorithmic}

Table 1. Compatible operations in the KVS interface.

<table>
<thead>
<tr>
<th>$T_{\text{AD}}(u, o)$</th>
<th>$\text{ad-put}(x, \cdot)$</th>
<th>$\text{ad-put}(y, \cdot)$</th>
<th>$\text{ad-get}(x)$</th>
<th>$\text{ad-get}(y)$</th>
<th>$\text{ad-del}(x)$</th>
<th>$\text{ad-del}(y)$</th>
<th>$\text{ad-list}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{ad-put}(x, \cdot)$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{ad-put}(y, \cdot)$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{ad-get}(x)$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{ad-get}(y)$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{ad-del}(x)$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{ad-del}(y)$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{ad-list}$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm 6 Implementation of VICOS at the client.

function vicos-put($k, v$)
|x| ← a random nonce
|cos-put($k||x, v$)
|h| ← hash($v$)
|r| ← aip-invoke(ad-put($k, (x, h)$))

if $r =$ ABORT then
|cos-del($k||x$) // concurrent incompatible operation
return $r$

function vicos-get($k$)
|r| ← aip-invoke(ad-get($k$))

if $r =$ ABORT then // concurrent incompatible operation
|return| ABORT

$(x, h)\leftarrow r$
|v| ← cos-get($k||x$)
|assert| hash($v$) = $h$
|return| $v$

function vicos-del($k$)
|r| ← aip-invoke(ad-del($k$))

if $r =$ ABORT then // concurrent incompatible operation
|return| ABORT
|cos-del($k||*$) // deletes all keys with prefix $k$
return $r$

function vicos-list()
|r| ← aip-invoke(ad-list())

if $r =$ ABORT then // concurrent incompatible operation
|return| ABORT
return $r$

4.4 Correctness

It is easy to see that the implementation of VICOS satisfies the two properties of a fork-linearizable Byzantine emulation. First, when $S$ is correct, then the clients proceed with their operations and all verification steps succeed. Hence, VICOS produces a linearizable execution. The linearization order is established by the operations AD-AIP. Furthermore, when the clients execute sequentially, then by the corresponding property of AIP, no client ever receives ABORT from AD-AIP.

Second, consider the case of a malicious server controlling COS and the AD-AIP server together. AIP ensures that the operations on AD-AIP (ad-put, ad-get, etc.) are fork-linearizable. The implementation of AD-AIP follows the known approach of memory checking with hash trees [4, 23] and therefore authenticates the object hash values that VICOS writes to AD-AIP. According to the properties of the hash function, the objects themselves are identified uniquely by their hash values. Since VICOS ensures the object data written to COS or returned to the client corresponds to the hash value stored in AD-AIP, it follows that all operations of VICOS are also fork-linearizable.
5 Evaluation

We have implemented VICOS in Java, using the BlobStore interface of Apache jclouds for connecting to different object stores. The AD-AIP server runs as a standalone web service, communicating with the client using Akka 2.3.7 (http://www.akka.io). The core implementation of the VICOS protocol consists of ~3400 sloc, the server part is ~400 sloc more, whereas the client part including the integration with the evaluation platform (see below) takes ~800 sloc extra.

5.1 Experimental setup

Our experiments use cloud servers and an OpenStack Swift-based object storage service (http://swift.openstack.org) of a major cloud provider with about two dozen data centers world-wide (Softlayer — an IBM Company, http://www.softlayer.com/object-storage).

The AD-AIP server runs on a dedicated VM with 4 × 2 GHz virtual CPUs and 4 GB RAM with 1 Gbps Ethernet. The clients run on 8 VMs with each 4 × 2 GHz virtual CPUs and 4 GB RAM with 100 Mbps network. All clients are hosted in the same data center. All machines run Ubuntu 14.04-64 Linux and Oracle Java (JRE 8, build 1.8.0_31-b13).

The implementation uses 2048 bit RSA-SHA1 signatures accessed using JCE. Microbenchmarks show that the signing operation on a client takes around 5 ms, while the verification takes around 250 µs, for inputs of size 1 kB to 100 kB.

To simulate a realistic environment, we conduct experiments in two settings. A datacenter setting, with all components in the same data center, and a wide-area setting, in which the server and the object store are located together in one data center (“Toronto”), and the clients at a remote site (“Dallas”).

The datacenter setting establishes a best-case baseline due to the very low network latencies (<1 ms). This deployment models only a corner case in terms of the security model because the clients and the storage service are co-located.

The wide-area setting exhibits a moderate network latency (~60 ms) between the two data centers and models the typical case of geographically distributed clients accessing a cloud service with its point-of-access on the same continent.

The evaluation is driven by COSBench [30] (version 0.4.0), an extensible tool for benchmarking cloud object stores. We have created an adapter to drive VICOS from COSBench.

COSBench uses a distributed architecture, consisting of multiple drivers, which generate the workload and simulate clients invoking a cloud object store, and one controller, which controls the drivers, selects the workload parameters, collects results, and outputs aggregated statistics. In particular, our COSBench setup for VICOS reports the operation latency, defined as the time that an operation takes to complete, and the throughput, defined as data rate between the client and the cloud storage service.

COSBench invokes “read” and “write” operations, implemented by get and put, respectively. Every reported data point involves read and write operations taken over a period of at least 90 s after a 90 s warm-up. For each experiment, two configurations are measured: (1) the baseline with direct, unprotected access from COSBench to cloud storage; and (2) VICOS, running all operations with verification.

5.2 Results

5.2.1 Object size

In our first experiment we study how the object size affects the latency and the throughput of VICOS. For this experiment we define a workload with a single client executing read and write operations for objects of size 1 kB, 10 kB, 100 kB, and 1 MB.
Figure 3. The effect of different object sizes: Latency and throughput of read and write operations with one client.

The graphs in Fig. 3 show that the latency and throughput characteristics of VICOS behave similar to the baseline. As expected, we observe that VICOS introduces an overhead that increases latency and decreases throughput. The overhead is is about the same for the smaller objects (1 kB and 10 kB) and disappears for the largest object size (1 MB). Note that the baseline and VICOS only saturate the network in the datacenter setting and with 1 MB objects.

5.2.2 Number of clients

We also study the scalability of the system by increasing the number of clients. The workload uses up to 32 clients (spread uniformly over the 8 VMs), each accessing a separate object with fixed size of 100 kB.

As Fig. 4 shows, the baseline throughput scales almost linearly. VICOS does not follow the same behavior: for more than 8 clients, its throughput remains almost constant and its latency grows faster than that of the baseline. The reason is that all request by the clients are handled by the AD-AIP server sequentially and thus it becomes the system’s bottleneck.

5.2.3 Concurrent operations

Finally, we investigate the effect of non-compatible concurrent operations. VICOS aborts an operation in case they are not compatible and the client has to try again later. The prototype implementation automatically retries until the operation has finished. Naturally, this results in higher latencies when conflicts occur. In contrast, ACOP aborts all non-commuting operations [8].
To evaluate this behavior we created a workload with eight clients and split them into two groups; one group only invokes read operations and the other invokes only write operations. Each client operates on its own set of objects, such that no client ever aborts due to a concurrent operation of another client. Each set contains two objects with a fixed size of 100 kB. Then we cause operations to abort by progressively intersecting the object sets of each group. That is, we run this experiment from no intersection in the object sets to 100 % intersection, where all clients of one group access the same objects as the other group. The higher the intersection, the more clients concurrently operate on the same objects, invoke non-compatible operations, and cause operations to abort.

Figure 5 shows an increase of the average latency for ACOP with increasing object intersection. In contrast, for VICOS we observe steady latency and throughput for write operations and a smaller increasing latency only for read operations. This is due to a smaller number of aborts as described in Table 1.

6 Related Work

Many previous systems providing data integrity rely on trusted components. Distributed file systems with cryptographic protection (e.g., FARSITE [1] or SiRiUS [13]) cannot provide the strong notion of integrity and consistency given by VICOS because they rely on trusted directory services for freshness. Several recent systems ensure data integrity with the help of additional trusted hardware, such as CATS [29], which offers accountability based on an immutable public publishing medium, or A2M [10], which assumes an append-only memory. Iris [26] relies on a trusted gateway appliance, which mediates
all requests between the clients and the untrusted cloud storage.

With only one client, the classic solution for memory checking by Blum et al. [4] provides data integrity through a hash tree and by storing its root at the client. Many systems have exemplified this approach for remote file systems and for cloud storage (e.g., Athos [15]).

With authenticated data structures [23, 21], the single-writer, multi-reader model of remote storage can be authenticated, assuming a trusted way to distribute authenticators from the writer. In practice, this approach is often taken for software distribution, where new releases are authenticated by broadcasting hash values of the packages over a mailing list. VICOS and AIP represent one way to generalize ADS to multiple writers.

In the multi-client model, Mazières et al. [22, 19] have introduced the notion of fork-linearizability and implemented SUNDR, the first system to guarantee fork-linearizable views to all clients. It detects integrity and consistency violations among all clients that “see” each other’s operations. As mentioned in Sec. 1, several systems have expanded this notion to different applications [12] and improved its efficiency [9]. Others have explored aborting operations [20] or introduced weak fork-linearizability in order to avoid blocking operations, such as FAUST [7] and Venus [25]. Furthermore, VICOS also reduces the communication overhead, since SUNDR, FAUST, and Venus all use messages of size $\Theta(n)$ with $n$ clients, whereas the message size in VICOS does not depend on $n$.

BST [28] and COP [8] guarantee fork-linearizability for arbitrary services run by a Byzantine server, not only for data storage, and support wait-freedom for commuting operations.

VICOS builds directly on COP, but improves the efficiency by avoiding the local state copies at clients and by reducing the computation and communication overhead. The main advantage is that
clients can remain offline between executing operations without stalling the protocol.

7 Conclusion

This paper has presented VICOS, a complete system for protecting the integrity and consistency of data outsourced to untrusted commodity cloud object stores. It presents multiple algorithmic improvements compared to existing work.

In contrast to existing solutions, VICOS works with commodity cloud storage services and ensures the best possible consistency notion of fork-linearizability. It supports wait-free client operations and does not require any additional trusted components.

There are several challenges that this paper not address, which remain open for future work. An interesting question, for instance, is how to recover from an integrity violation. Since we assume only a single untrusted server and that the clients’ data reside at the cloud storage service, this seems not trivial.

Another interesting challenge is a step towards a more realistic system model where we also consider malicious clients. For small groups of clients our system model makes sense, but for groups with hundreds of clients we can not maintain this assumption. The situation is especially interesting when a client colludes with the malicious server.

A more specific improvement would optimize the integrity protection protocol, in particular, one may reduce communication overhead for operations that do not affect the server’s state, such as read operations. It seems possible to skip the passive phase in this case.

Finally, the approach of AIP also be applied to services beyond cloud storage; for example, cloud and NoSQL databases, interactions in a social network, or certificate and key management services.

Acknowledgments

We thank Rüdiger Kapitza, Dieter Sommer and Sören Bleikertz for interesting discussions and helpful comments.

References


