Verifying the Consistency of Remote Untrusted Services with Commutative Operations

Christian Cachin  
IBM Research - Zurich  
cca@zurich.ibm.com

Olga Ohrimenko*  
Brown University  
olya@cs.brown.edu

9 December 2013

Abstract

A group of mutually trusting clients outsources a computation service to a remote server, which they do not fully trust and that may be subject to attacks. The clients do not communicate with each other and would like to verify the correctness of the remote computation and the consistency of the server’s responses. This paper presents the Commutative-Operation verification Protocol (COP) that ensures linearizability when the server is correct and preserves fork-linearizability in any other case. Fork-linearizability ensures that all clients that observe each other’s operations are consistent in the sense that their own operations and those operations of other clients that they see are linearizable. COP strictly generalizes previous protocols by modeling abortable services and by supporting wait-free client operations for sequences of arbitrary commutative operations. When combined with authenticated data structures, COP also adds distributed consistency verification to computations with outsourced state, where the clients do not store the whole state of the computation.

1 Introduction

With the advent of cloud computing, most computations run in remote data centers and no longer on local devices. As a result, users are bound to trust the service provider for the confidentiality and the correctness of their computations. This work addresses the integrity of outsourced data and computations and the consistency of the provider’s responses. Consider a group of mutually trusting clients who want to collaborate on a resource that is provided by a remote partially trusted server. This could be a wiki containing data of a common project, an archival document repository, or a groupware tool running in the cloud. A subtle change in the remote computation, whether caused inadvertently by a bug or deliberately by a malicious adversary, may result in wrong responses to the clients. The clients trust the provider only partially, hence, they would like to assess the integrity of the computation, to verify that responses are correct, and to check that they all get consistent responses.

In an asynchronous network model without communication among clients such as considered here, the server may perform a forking attack and omit the effects of operations by some clients in her responses to other clients. Not knowing which operations other clients execute, the latter group cannot detect such violations. The best achievable consistency guarantee in this setting is captured by fork-linearizability, introduced by Mazieres and Shasha [20] for storage systems. Fork-linearizability ensures that whenever the server in her responses to a client $C_1$ has ignored a write operation executed by a client $C_2$, then $C_1$ can never again read a value written by $C_2$ afterwards and vice versa. From this property, clients can detect server misbehavior from a single inconsistent operation, which is much easier than comparing the effects of all past operations one-by-one.
Several conceptual [6, 18, 4, 5] and practical advances [27, 8, 17, 23] have recently been made that improve consistency checking and verification with fork-linearizability and related notions for remote storage and computation. The resulting protocols ensure that when the server is correct, the service is linearizable and (ideally) the algorithm is wait-free, that is, every client’s operations complete independently of other clients. It has been recognized, however, that read/write conflicts often cause such protocols to block; this applies to consistency verification for storage with fork-linearizable semantics [20, 6] and for other forking consistency notions [4, 5].

In this paper, we go beyond storage services and propose a new protocol for consistency verification of remote computations on a Byzantine server. The Commutative-Operation verification Protocol or COP supports arbitrary functionalities, exploits commuting operations, and allows clients to operate concurrently and without blocking or aborting whenever feasible, while imposing fork-linearizable semantics. Through this guarantee Byzantine behavior of the server can be exposed easily. For instance, the clients can exchange some simple status messages out-of-band to check that the server operated correctly. Clients may therefore verify the correctness of a service in an end-to-end way.

Support for wait-free operations is a key feature for collaboration with remote coordination, as geographically separated clients may operate with totally different timing characteristics. Consequently, previous work has devoted a lot of attention to identifying and avoiding blocking situations [20, 6, 15]. For example, read operations in a storage service commute and do not lead to a conflict. On the other hand, when a client writes to a data item concurrently with another client reading from the item, the reader has to wait until the write operation completes; otherwise, fork-linearizability is not guaranteed. If all operations are to proceed without blocking, though, it is necessary to weaken the consistency guarantees to fork-* linearizability [13] or weak fork-linearizability [5], for instance. COP is wait-free and never blocks because it aborts non-commuting operations that cannot proceed. Abortable operations have been introduced in this context by Majuntke et al. [18].

The Blind Stone Tablet (BST) protocol [27], the closest predecessor of this work, supports an encrypted remote database hosted by an untrusted server that is accessed by multiple clients. Its consistency checking algorithm allows some commuting client operations to proceed concurrently, but only to a limited extent, as we explain below. Furthermore, the BST protocol guarantees fork-linearizability for database state updates, but does not ensure it for certain responses output by a client.

SPORC [8] considers a groupware collaboration service whose operations may not commute, but can be made to commute by applying operational transformations. Through this mechanism, different execution orders still converge to the same state. All SPORC operations are wait-free and respect fork-* linearizability.

### 1.1 Contributions

This paper considers a generic service executed by an untrusted server and provides a new protocol for consistency verification through fork-linearizable semantics. It explores the relation between commuting operations in the service specification and client operations that may proceed concurrently.

More concretely, this paper introduces the Commutative-Operation verification Protocol (COP) and makes four contributions:

1. COP is the first wait-free, abortable protocol that emulates an arbitrary functionality on a Byzantine server with fork-linearizability and supports commuting operation sequences.
2. COP allows clients to proceed at their own speed, regardless of the behavior of other clients, when they execute non-conflicting sequences of operations.
3. We formally prove COP correct and demonstrate that all completed operations and their responses respect fork-linearizability.
4. We describe how to combine COP with an authenticated data structure in order to provide authenticated remote computation.

COP follows the general pattern of most previous fork-linearizable emulation protocols. For determining when to proceed with concurrent operations, it considers sequences of operations that jointly commute, in contrast to earlier protocols, which considered only isolated operations.

In basic COP, the server merely coordinates client-side operations but does not compute the results. This conceptually simple approach can be found in many related protocols and practical collaboration systems. It also represents the common trend of cloud computing to shift computation to the client and coordination to the cloud. In the extension of COP to authenticated remote computation, however, operations are executed by the server and the clients may not know the complete state of the computation. This enables COP to handle services with large state, such as storage systems.

1.2 Related work

1.2.1 Storage protocols

Fork-linearizability has been introduced (under the name of fork consistency) together with the SUNDR storage system. Conceptually SUNDR operates on storage objects with simple read/write semantics. Subsequent work of Cachin et al. improves the efficiency of untrusted storage protocols. A lock-free storage protocol with abortable operations, which lets all operations complete in the absence of step contention, has been proposed by Majuntke et al.

FAUST and Venus go beyond the fork-linearizable consistency guarantee and model occasional message exchanges among the clients. This allows FAUST and Venus to obtain stronger semantics, in the sense that they eventually reach consistency (i.e., linearizability) or detect server misbehavior. In the model considered here, fork-linearizability is the best possible guarantee.

1.2.2 Blind Stone Tablet (BST)

The BST protocol considers transactions on a common database, coordinated by the remote server. Clients first simulate a transaction on their own copy, potentially generating local output, then coordinate with the server for ordering the transaction. From the server’s response the client determines if his transaction commutes with other, pending transactions invoked by different clients that were reported by the server. If they conflict, the client undoes the transaction and basically aborts; otherwise, he commits the transaction and relays it via the server to other clients. When a client receives such a relayed transaction, the client applies the transaction to its database copy.

BST has two limitations: First, because a client applies his own transactions only when all pending transactions by other clients have been applied to his own state, updates induced by his transactions are delayed in dependence on other clients. Thus, he cannot always execute his next transaction from the modified state and obtain a correct output. This implies the client is blocked, actually. Second, the notion of “trace consistency” in the analysis of the BST protocol considers only transactions that have been applied to the local state, not local output generated by the client. Hence, fork-linearizability is not shown for the service responses but only for those transactions that clients have applied to their state (the former may occur long before the latter). In contrast, COP guarantees fork-linearizability for all responses output by clients.
COP is strictly more general than BST, as it allows one client to execute multiple operations independently of the other clients, as long as his sequence of operations jointly commutes with the sequence of pending operations by other clients. BST considers only the commutativity of individual operations. Note that two operations $o_1$ and $o_2$ may independently commute with an operation $o_3$ from a particular starting state, but their concatenation, $o_1 \circ o_2$, may not commute with $o_3$ (see the example in Section 3).

1.2.3 Non-blocking protocols

SPORC [8] is a group collaboration system where operations do not need to be executed in the same order at every client by virtue of employing operational transforms. The latter concept allows for shifting operations to a different position in an execution by transforming them according to properties of the skipped operations. Differently ordered and transformed variants of a common sequence converge to the same end state. SPORC achieves fork-*$ linearizability [15], which is closely related to weak fork-linearizability [5]; both notions are relaxations of fork-linearizability that permit concurrent operations to proceed without blocking, such that protocols become wait-free. The increased concurrency is traded for weaker consistency, as up to one diverging operation may exist between the views of different clients and cannot be detected. FAUST [5], mentioned before, never blocks clients and enjoys eventual consistency, but guarantees only weak fork-linearizability.

In contrast to the SPORC and FAUST protocols, COP ensures the stronger fork-linearizability condition, where every operation is consistent as soon as it completes. SPORC is neither weaker nor stronger than COP: On one hand, SPORC seems more general as it never blocks clients even for operations that do not appear to commute; on the other hand, though, SPORC only supports functions with suitably transformable operations and has no provisions for handling conflicting operations, whereas COP works for arbitrary functions.

In all above protocols for generic services (BST, SPORC, and COP), all clients execute all operations. As suggested by Cachin [3], techniques for “authenticated separated execution” could eliminate this drawback; these methods extend the notion of authenticated data structures. For storage protocols (SUNDR and FAUST), executing other client’s operations is already not necessary because their functionality permits that.

1.3 Organization of the paper

The paper continues by introducing the notation and basic concepts in Section 2. The subsequent section presents COP and discusses its properties. An extension of COP to remote authenticated computation is described in Section 4. For lack of space, some definitions are contained in Appendix A, and the formal analysis that COP emulates an arbitrary functionality on a Byzantine server with fork-linearizability is presented in Appendix B.

2 Definitions

System model. We consider an asynchronous distributed system with $n$ clients, $C_1, \ldots, C_n$, and a server $S$, modeled as processes. Each client is connected to the server through an asynchronous, reliable communication channel that respects FIFO order. A protocol specifies the operations of the processes. All clients are correct and follow the protocol, whereas $S$ operates in one of two modes: either she is correct and follows the protocol or she is Byzantine and may deviate arbitrarily from the specification.

Functionality. We consider a deterministic functionality $F$ (also called a type) defined over a set of states $S$ and a set of operations $O$. $F$ takes as arguments a state $s \in S$ and an operation $o \in O$ and
returns a tuple \((s', r)\), where \(s' \in S\) is a state that reflects any changes that \(o\) caused to \(s\) and \(r \in R\) is a response to \(o\):

\[
(s', r) \leftarrow F(s, o).
\]

This is also called the sequential specification of \(F\).

We extend this notation for executing a sequence of operations \(\langle o_1, \ldots, o_k \rangle\), starting from an initial state \(s_0\), and write

\[
(s', r) = F(s_0, \langle o_1, \ldots, o_k \rangle)
\]

for \((s_i, r_i) = F(s_{i-1}, o_i)\) with \(i = 1, \ldots, k\) and \((s', r) = F(s_k, r_k)\). Note that an operation in \(O\) may represent a batch of multiple application-level operations.

Commutative Operations. Commutative operations of \(F\) play a role in protocols that may execute multiple operations concurrently. Two operations \(o_1, o_2 \in O\) are said to commute in a state \(s\) if and only if these operations, when applied in different orders starting from \(s\), yield the same respective states and responses. Formally, if

\[
\begin{align*}
(s', r_1) &\leftarrow F(s, o_1), & (s'', r_2) &\leftarrow F(s', o_2); & \text{and} & &
(t', q_2) &\leftarrow F(s, o_2), & (t'', q_1) &\leftarrow F(t', o_1)
\end{align*}
\]

then

\[
r_1 = q_1, \ r_2 = q_2, \ s'' = t''.
\]

Furthermore, we say two operations \(o_1, o_2 \in O\) commute when they commute in any state of \(S\).

Also sequences of operations can commute. Suppose two sequences \(\rho_1\) and \(\rho_2\) consisting of operations in \(O\) are mixed together into one sequence \(\pi\) such that the partial order among the operations from \(\rho_1\) and from \(\rho_2\) is retained in \(\pi\), respectively. If executing \(\pi\) starting from a state \(s\) gives the same respective responses and the same final state as for every other such mixed sequence, in particular for \(\rho_1 \circ \rho_2\) and for \(\rho_2 \circ \rho_1\), where \(\circ\) denotes concatenation, we say that \(\rho_1\) and \(\rho_2\) commute in state \(s\). Analogously, we say that \(\rho_1\) and \(\rho_2\) commute if they commute in any state.

Operations that do not commute are said to conflict. Commuting operations are well-known from the study of concurrency control [25][26]. They can be defined alternatively by considering only the responses of future operations and ignoring the state, but when allowing arbitrary functionalities \(F\), this notion is equivalent to ours, as \(F\) might contain an operation that returns the complete state. We define a Boolean predicate \(\text{commute}_F(s, \rho_1, \rho_2)\) that is true if and only if \(\rho_1\) and \(\rho_2\) commute in \(s\) according to \(F\).

Abortable services. When operations of \(F\) conflict, a protocol may either decide to block or to abort. Aborting and giving the client a chance to retry the operation at his own rate often has advantages compared to blocking, which might delay an application in unexpected ways.

As in previous work that permitted aborts [1][13], we allow operations to abort and augment \(F\) to an abortable functionality \(F'\) accordingly. \(F'\) is defined over the same set of states \(S\) and operations \(O\) as \(F\), but returns a tuple defined over \(S\) and \(R \cup \{\perp\}\). \(F'\) may return the same output as \(F\), but \(F'\) may also return \(\perp\) and leave the state unchanged, denoting that a client is not able to execute \(F\). Hence, \(F'\) is a non-deterministic relation and satisfies

\[
F'(s, o) = \{(s, \perp), (s, o)\}.
\]

Since \(F'\) is not deterministic, a sequence of operations no longer uniquely determines the resulting state and response value.

Abortable functionalities are related to obstruction-free objects [1][12], which guarantee that every client operation completes assuming the client eventually runs in isolation.
Operations and histories. The clients interact with $F$ through operations provided by $F$. As operations take time, they are represented by two events occurring at the client, an invocation and a response. A history of an execution $\sigma$ consists of the sequence of invocations and responses of $F$ occurring in $\sigma$. An operation is complete in a history if it has a matching response.

An operation $o$ precedes another operation $o'$ in a sequence of events $\sigma$, denoted $o <_\sigma o'$, whenever $o$ completes before $o'$ is invoked in $\sigma$. A sequence of events $\pi$ preserves the real-time order of a history $\sigma$ if for every two operations $o$ and $o'$ in $\pi$, if $o <_\sigma o'$ then $o <_\pi o'$. Two operations are concurrent if neither one of them precedes the other. A sequence of events is sequential if it does not contain concurrent operations. For a sequence of events $\sigma$, the subsequence of $\sigma$ consisting only of events occurring at client $C_i$ is denoted by $\sigma|C_i$ (we use the symbol $|$ as a projection operator). For some operation $o$, the prefix of $\sigma$ that ends with the last event of $o$ is denoted by $\sigma[0]$. An operation $o$ is said to be contained in a sequence of events $\sigma$, denoted $o \in \sigma$, whenever at least one event of $o$ is in $\sigma$. We often simplify the terminology by exploiting that every sequential sequence of events corresponds naturally to a sequence of operations, and that analogously every sequence of operations corresponds to a sequential sequence of events.

An execution is well-formed if the events at each client are alternating invocations and matching responses, starting with an invocation. An execution is fair, informally, if it does not halt prematurely when there are still steps to be taken or messages to be delivered (see the standard literature for a formal definition [16]). We are interested in a protocol where the clients never block, though some operations may be aborted and thus will not complete regularly. We call a protocol wait-free if in every history where the server is correct, every operation by any client completes [11].

Consistency properties. Clients interact with $F$ via operations. Recall that every operation at a client $C_i$ is associated with an invocation and a response event that occurs at $C_i$. We say that $C_i$ executes an operation between the corresponding invocation and completion events.

We use the standard notion of linearizability [13], which requires that the operations of all clients appear to execute atomically in one sequence, and its extension to fork-linearizability [20, 6], which relaxes the condition of one sequence to permit multiple “forks” of an execution. Under fork-linearizability, every client observes a linearizable history and when some operation is observed by multiple clients, the history of events up to this operation is the same.

Our protocol provides a fork-linearizable Byzantine emulation [6] of the service on an untrusted server. This notion ensures two dual properties: first, when the server is correct, then the service should guarantee the standard notion of linearizability; otherwise, the protocol should ensure fork-linearizability to the clients. Formal definitions appear in Appendix A.1.

Cryptography. We make use of two cryptographic primitives, namely a collision-free hash function $\text{hash}$ and a digital signature scheme, with operations denoted by $\text{sign}_i$ and $\text{verify}_i$ for signatures computed by $C_i$. As our focus lies on concurrency and correctness and not on cryptography, we model both as ideal, deterministic functionalities implemented by a trusted entity. Details are stated in Appendix A.2.

3 The commutative-operation verification protocol

The pseudocode of COP for the clients and the server is presented in Algorithms 1–3. We assume that the execution of each client is well-formed and fair.
Notation. The function \( \text{length}(a) \) for a list \( a \) denotes the number of elements in \( a \). Several variables are dynamic arrays or maps, which associate keys to values. A value is stored in a map \( H \) by assigning it to a key, denoted \( H[k] \leftarrow v \); if no value has been assigned to a key, the map returns \( \bot \). Recall that \( F' \) is the abortable extension of functionality \( F \).

Overview. COP adopts the structure of previous protocols that guarantee fork-linearizable semantics \([20, 27, 3]\). It aims at obtaining a globally consistent order for the operations of all clients, as determined by the server.

When a client \( C_i \) invokes an operation \( o \), he sends an INVOKE message to the server \( S \). He expects to receive a REPLY message from \( S \) telling him about the position of \( o \) in the global sequence of operations. The message contains the operations that are pending for \( o \), that is, operations that \( C_i \) may not yet know and that are ordered before \( o \) by \( S \). We distinguish between pending-other operations invoked by other clients and pending-self operations, which are operations executed by \( C_i \) up to \( o \).

Client \( C_i \) then verifies that the data from the server is consistent. In order to ensure fork-linearizability for his response values, the client first simulates the pending-self operations and tests if \( o \) commutes with the pending-other operations. If the test succeeds, he declares \( o \) to be successful, executes \( o \), and computes the response \( r \) according to \( F \); otherwise, \( O \) is aborted and the response is \( r = \bot \). According to this, the status of \( o \) is a value in \( Z = \{\text{SUCCESS}, \text{ABORT}\} \). Through these steps the client commits \( o \). Then he sends a corresponding COMMIT message to \( S \) and outputs \( r \).

The (correct) server records the committed operation and relays it to all clients via a BROADCAST message. When the client receives such a broadcasted operation, he verifies that it is consistent with everything the server told him so far. If this verification succeeds, we say that the client confirms the operation. If the operation’s status was SUCCESS, then the client executes it and applies it to his local state.

Data structures. Every client locally maintains a set of variables during the protocol. The state \( s \in S \) is the result of applying all successful operations, received in BROADCAST messages, to the initial state \( s_0 \). Variable \( c \) stores the sequence number of the last operation that the client has confirmed. \( H \) is a map containing a hash chain computed over the global operation sequence as announced by \( S \). The contents of \( H \) are indexed by the sequence number of the operations. Entry \( H[l] \) is computed as \( \text{hash}(H[l − 1 || o || l || i]) \), with \( H[0] = \text{null} \), and represents an operation \( o \) with sequence number \( l \) executed by \( C_i \). (The notation \( || \) stands for concatenating values as bit strings.) A variable \( u \) is set to \( o \) whenever the client has invoked an operation \( o \) but not yet completed it; otherwise \( u \) is \( \bot \). Variable \( Z \) maps the sequence number of every operation that the client has executed himself to the status of the operation. The client only needs the entries in \( Z \) with index greater than \( c \).

The server also keeps several variables locally. She stores the invoked operations in a map \( I \) and the completed operations in a map \( O \), both indexed by the operations’ sequence numbers. Variable \( t \) determines the global sequence number for the invoked operations. Finally, variable \( b \) is the sequence number of the last broadcasted operation and ensures that \( S \) disseminates operations to clients in the global order.

Protocol. When client \( C_i \) invokes an operation \( o \), he stores it in \( u \) and sends an INVOKE message to \( S \) containing \( o, c, \) and \( \tau \), a digital signature computed over \( o \) and \( i \). In turn, a correct \( S \) sends a REPLY message with the list \( \omega \) of pending operations; they have a sequence number greater than \( c \). Upon receiving a REPLY message, the client checks that \( \omega \) is consistent with any previously sent operations and uses \( \omega \) to assemble the successful pending-self operations \( \mu \) and the pending-other operations \( \gamma \). He then determines whether \( o \) can be executed or has to be aborted.
Algorithm 1 Commutative-operation verification protocol (client $C_i$)

State

$u \in \mathcal{O} \cup \{\bot\}$: the operation being executed currently or $\bot$ if no operation runs, initially $\bot$
$c \in \mathbb{N}_0$: sequence number of the last operation that has been confirmed, initially 0
$H : \mathbb{N}_0 \to \{0, 1\}^*$: hash chain (see text), initially containing only $H[0] = \text{NULL}$
$Z : \mathbb{N}_0 \to \mathbb{Z}$: status map (see text), initially empty
$s \in \mathcal{S}$: current state, after applying operations, initially $s_0$

upon invocation $o$ do

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

upon receiving message $[\text{REPLY}, \omega]$ from $S$ do

$\gamma \leftarrow \langle \rangle$ // list of pending-other operations
$\mu \leftarrow \langle \rangle$ // list of successful pending-self operations
$k \leftarrow 1$
while $k \leq \text{length}(\omega)$ do

$(o, j, \tau) \leftarrow \omega[k]$ // promised sequence number of $o$
$l \leftarrow c + k$
if not $\text{verify}_j(\tau, \text{INVOKE}||o||j)$ then

halt
if $H[l] = \bot$ then

$H[l] \leftarrow \text{hash}(H[l-1]||o||l||j)$ // extend hash chain
else if $H[l] \neq \text{hash}(H[l-1]||o||l||j)$ then

halt // server replies are inconsistent
if $j = i \land Z[l] = \text{SUCCESS}$ then

$\mu \leftarrow \mu \circ (o)$
ext if $j \neq i$ then

$\gamma \leftarrow \gamma \circ (o)$
$k \leftarrow k + 1$
if $k = 1 \lor o \neq u \lor j \neq i$ then

halt // last pending operation must equal the current operation
$(a, r) \leftarrow F(s, \mu)$ // compute temporary state with successful pending-self operations
if $\text{commute}_F(a, \langle u \rangle, \gamma)$ then

$(a, r) \leftarrow F(a, u)$
$Z[l] \leftarrow \text{SUCCESS}$
else

$r \leftarrow \bot$
$Z[l] \leftarrow \text{ABORT}$
$\phi \leftarrow \text{sign}_i(\text{COMM}||u||l||H[l]||Z[l])$
send message $[\text{COMM}, u, l, H[l], Z[l], \phi]$ to $S$
$u \leftarrow \bot$
return $r$
Algorithm 2 Commutative-operation verification protocol (client $C_i$, continued)

upon receiving message $[\text{broadcast}, o, q, h, z, \phi, j]$ from $S$ do  
   if not $(q = c + 1$ and verify$_j(\phi, \text{commit}\|o\|q\|h\|z))$ then  
      halt  /* server replies are not consistent */  
   if $H[q] = \bot$ then  
      $H[q] \leftarrow \text{hash}(H[q - 1]\|o\|q\|j)$  /* operation has not been pending at client */  
   if $h \neq H[q]$ then  
      halt  /* server replies are not consistent, operation is not confirmed */  
   if $z = \text{SUCCESS}$ then  
      $(s, r) \leftarrow F(s, o)$  // apply the operation and ignore response  
   $c \leftarrow c + 1$

Algorithm 3 Commutative-operation verification protocol (server $S$)

State  
$t \in \mathbb{N}_0$: sequence number of the last invoked operation, initially 0  
$b \in \mathbb{N}_0$: sequence number of the last broadcasted operation, initially 0  
$I : \mathbb{N} \to O \times \mathbb{N}_0 \times \{0, 1\}^*$: invoked operations (see text), initially empty  
$O : \mathbb{N} \to O \times \{0, 1\}^* \times Z \times \{0, 1\}^* \times \mathbb{N}$: committed operations (see text), initially empty

upon receiving message $[\text{invoke}, o, c, \tau]$ from $C_i$ do  
   $t \leftarrow t + 1$  
   $I[t] \leftarrow (o, i, \tau)$  
   $\omega \leftarrow \langle I[c + 1], I[c + 2], \ldots, I[t] \rangle$  // include non-committed operations and $o$  
   send message $[\text{reply}, \omega]$ to $C_i$

upon receiving message $[\text{commit}, o, q, h, z, \phi]$ from $C_i$ do  
   $O[q] \leftarrow (o, h, z, \phi, i)$  
   while $O[b + 1] \neq \bot$ do  // broadcast operations ordered by their sequence number  
      $b \leftarrow b + 1$  
      $(o', h', z', \phi', j) \leftarrow O[b]$  
      send message $[\text{broadcast}, o', b, h', z', \phi', j]$ to all clients

In particular, during the loop in Algorithm 1 for every operation $o$ in $\omega$, $C_i$ determines its sequence number $l$ and verifies that $o$ was indeed invoked by $C_j$ from the digital signature. He computes the entry of $o$ in the hash chain from $o$ itself, $l$, $j$, and $H[l - 1]$. If $H[l] = \bot$, then $C_i$ stores the hash value there. Otherwise, if $H[l]$ has already been set, $C_i$ verifies that the hash values are equal; this means that $o$ is consistent with the pending operation(s) that $S$ has sent previously with indices up to $l$.

If operation $o$ is his own and its saved status in $Z[l]$ was SUCCESS, then he appends it to $\mu$. The client remembers the status of his own operations in $Z$, since $\text{commute}_\phi$ depends on the state and that could have changed if he applied operations after committing $o$.

Finally, when $C_i$ reaches the end of $\omega$ (i.e., when $C_i$ considers $o = u$), he checks that $\omega$ is not empty and that it contains $u$ at the last position. He then creates a temporary state $a$ by applying $\mu$ to the current state $s$, and tests whether $u$ commutes with the pending-other operations $\gamma$ in $a$. If they do, he records the status of $u$ as SUCCESS in $Z[l]$ and computes the response $r$ by executing $u$ on state $a$. If $u$ does not
commute with $\gamma$, he sets status of $u$ to ABORT and $r \leftarrow \perp$. Then $C_i$ signs $u$ together with its sequence number, status, and hash chain entry $H[l]$ and includes all values in the COMMIT message sent to $S$.

Upon receiving a COMMIT message for an operation $o$ with a sequence number $q$, the (correct) server records its content as $O[q]$ in the map of committed operations. Then she is supposed to send a BROADCAST message containing $O[q]$ to the clients. She waits with this until she has received COMMIT messages for all operations with sequence number less than $q$ and broadcasted them. This ensures that completed operations are disseminated in the global order to all clients. Note that this does not forbid clients from progressing with their own operations as we explain below.

In a BROADCAST message received by client $C_i$, the committed operation is represented by a tuple $(o, q, h, z, \phi, j)$. He conducts several verification steps before confirming the operation $o$ and applying it to his state $s$. First, he verifies that the sequence number $q$ is the next operation according to his variable $c$, hence, $o$ follows the global order and the server did not omit any operations. Second, he uses the digital signature $\phi$ on the information in the message to verify that the client $C_j$ indeed committed $o$. Lastly, the client computes his own hash-chain entry $H[q]$ for $o$ and confirms that it is equal to the hash-chain value $h$ from the message. This ensures that $C_i$ and $C_j$ have received consistent operations from $S$ up to $o$. Once the verification succeeds, the client applies $o$ to his state $s$ only if its status $z$ was SUCCESS, that is, when $C_j$ has not aborted $o$.

**Commuting operation sequences.** Consider the following example $F$ of a counter restricted to non-negative values: Its state consists of an integer $s$; an $\text{add}(x)$ operation adds $x$ to $s$ and returns TRUE; a $\text{dec}(x)$ operation subtracts $x$ from $s$ and returns TRUE if $x \geq s$, but does nothing and returns FALSE if $x < s$. Suppose the current state $s$ at $C_i$ is 7 and $C_i$ executes $\text{dec}(4)$ and subsequently $\text{dec}(6)$. During both operations of $C_i$, the server announces that $\text{add}(2)$ by another client is pending. Note that $C_i$ executes $\text{dec}(4)$ successfully but aborts $\text{dec}(6)$ because $\text{dec}(6)$ does not commute with $\text{add}(2)$ from 3, the temporary state (denoted by $c$ in Algorithm [1]) computed by $C_i$ after the pending-self operation. However, the latter two operations, $\text{add}(2)$ and $\text{dec}(6)$, do commute in the current state 7. This shows why the client must execute the pending-self operations before testing the current operation for a conflict.

Suppose now the current state $c$ is again 7 and $C_i$ executes $\text{dec}(4)$. The server reports the pending sequence $\langle \text{dec}(1), \text{dec}(3) \rangle$. Thus, $C_i$ aborts $\text{dec}(4)$. Even though $\text{dec}(4)$ commutes with $\text{dec}(1)$ and with $\text{dec}(3)$ individually in state 7, it does not commute with their sequence. This illustrates why one must check for a conflict with the sequence of pending operations.

**Memory requirements.** For saving storage space, the client may garbage-collect entries of $H$ and $Z$ with sequence numbers smaller than $c$. The server can also save space by removing the entries in $I$ and $O$ for the operations that she has broadcast. However, if new clients are allowed to enter the protocol, the server should keep all operations in $O$ and broadcast them to new clients upon their arrival.

With the above optimizations the client has to keep only pending operations in $H$ and pending-self operations in $Z$. The same holds for the server: the maximum number of entries stored in $I$ and $O$ is proportional to the number of pending operations at any client.

**Communication.** Every operation executed by a client requires him to perform one roundtrip to the server: send an INVOKE message and receive a REPLY. For every executed operation the server simply sends a BROADCAST message. Clients do not communicate with each other in the protocol. However, as soon as they do, they benefit from fork-linearizability and can easily discover a forking attack by comparing their hash chains.

Messages INVOKE, COMMIT and BROADCAST are independent of the number of clients and contain only a description of one operation, while the REPLY message contains the list of pending operations $\omega$. 

10
If even one client is slow, then the length of $\omega$ for all other clients grows proportionally to the number of further operations they are executing. To reduce the size of REPLY messages, the client can remember all pending operations received from $S$, and $S$ can send every pending operation only once.

**Aborts and wait-freedom.** Every client executing COP can proceed with an operation $o$ for $F$ as long as it does not conflict with pending operations of other clients. Observe that the state used by the client for executing $o$ reflects all of his own operations executed so far, even if he has not yet confirmed or applied them to his state because operations of other clients have not yet completed. After successfully executing $o$, the client outputs the response immediately after receiving the REPLY message from $S$. A conflict arises when $o$ does not commute with the pending operations of other clients. In this case, the client aborts $o$ and outputs $\bot$, according to $F'$.

Hence, for $F$ where all operations and operation sequences commute, COP is wait-free. For arbitrary $F$, however, no fork-linearizable Byzantine emulation can be wait-free [6]. COP permits aborts via the augmented functionality $F'$ in order to avoid blocking. As detailed before, every client completes every operation in the sense of $F'$, which includes aborts; therefore, COP is wait-free for $F'$. In other words, regardless of whether an operation aborts or not, the client may proceed executing further operations.

To mitigate the risk of conflicts, the clients may employ a synchronization mechanism such as a contention manager, scheduler, or a simple random waiting strategy. This kind of synchronization is less typical for storage systems but quite common for services with higher consistency demands. If one considers also clients that may crash (outside our formal model), then the client group has to be adjusted dynamically or a single crashed client might hold up progress of other clients forever. Previous work on the topic has explored how a group manager or a peer-to-peer protocol may control a group membership protocol [14, 23].

**Analysis.** COP emulates the abortable functionality $F'$ on a Byzantine server with fork-linearizability. Furthermore, all histories of COP where the clients execute operations sequentially are fork-linearizable w.r.t. $F$ (no operations abort), and if, additionally, the server is correct, then all such histories are also linearizable w.r.t. $F$. Here we give only a brief summary of this result; for lack of space the details appear in Appendix B.

There are two points to consider. First, with a correct $S$, we show that the output of every client satisfies $F'$ also in the presence of many pending-self operations. The check for commutativity, applied after simulating the client’s pending-self operations, ensures that the client’s response is the same as if the pending-other operations would have been executed before the operation itself.

The second main innovation lies in the construction of a view for every client that includes all operations that he has executed or applied, together with those of his operations that some other clients have confirmed. Since these operations may have changed the state at other clients, they must be considered. More precisely, some $C_k$ may have confirmed an operation $o$ executed by $C_i$ that $C_i$ has not yet confirmed or applied. In order to be fork-linearizable, the view of $C_i$ must include $o$ as well, including all operations that were “promised” to $C_i$ by $S$ in the sense that they were announced by $S$ as pending for $o$. It follows from the properties of the hash chain that the view of $C_k$ up to $o$ is the same as $C_i$’s view including the promised operations. The view of $C_i$ further includes all operations that $C_i$ has executed after $o$. Taken together this demonstrates that every execution of COP is fork-linearizable w.r.t. $F'$. 


4 Authenticated Computation

As introduced above, the COP server merely coordinates client-side operations but does not compute any responses nor relieve the clients from storing, in principle, the complete state of the service. This is also the approach taken by the BST protocol and other related systems \[27, 10, 8\]. As argued before, many common collaboration tools and cloud applications use this model.

In this section we consider the alternative model of authenticated remote computation, where only the server maintains the complete state, performs the bulk of the computation by herself, and sends results to clients. For example, storage systems with complex interfaces, databases, and application servers fall into this model. A blueprint for remote services with authentication has been proposed by Cachin \[3\]. Verification of such a service is possible if the computation can be authenticated and verified by the client without executing the whole computation himself in a so-called “authenticated separate execution.” The sever of this system has to augment her responses with an authenticator, which lets clients verify that the computation was performed correctly. In the following we describe how to extend COP for remote authenticated computation using an authenticated data structure. Such authentication methods are known for many data storage, indexing, and retrieval schemes, with typical operations that permit updates and queries. Clients accessing this service can verify the consistency and integrity of their interactions. Our approach may be applied to any outsourced computation for which the server can provide an authenticator. Recent advances in cryptographic tools for verifying remote computation suggest that it may even become feasible to construct authenticators for generic computations \[9, 2\].

4.1 Authenticated Data Structures

We consider a server that stores shared data. The clients access the data by invoking potentially complex storage and retrieval operations. The server applies operations to the data and sends back responses. To verify the integrity of the responses we use an authenticated data structure (ADS) \[21\], where the server stores $D$ together with authentication information $A$ for every operation index $b$. An execution of an operation, $(D, A, o) \leftarrow \text{exec}(D, o)$, is augmented by authentication information $A$ and becomes:

$$(D', A', r) \leftarrow \text{authexec}(D, A, o).$$

The server sends the response $r$ together with $A'$ to the client. The client maintains an authentication value or digest $d$ between operations and verifies the integrity of $r$ by computing:

$$(d, r') \leftarrow \text{verify}(d, A', o, r).$$

If $r' = \perp$, then the verification failed, otherwise the response $r' = r$ is valid. For practical authentication techniques such as hash trees and authenticated dictionaries, $A$ is usually much smaller than $D$ \[24, 19\].

We now describe how to extend COP from the client-centric approach in Algorithms \[1\]-\[3\] to the model where the server maintains the state.

4.2 Server

We start by describing the changes to Algorithm \[3\] for $S$. As part of her state $S$ additionally maintains a map $D : \mathbb{N}_0 \to \{0, 1\}^*$ and authentication information $A$ for every operation index $b$, where $D[0] = D_0$.
is the initial state. The entry $D[b]$ is assigned when the server broadcasts an operation with sequence number $b$. She also stores $D[q]$ for past sequence numbers $q = 0, \ldots, b - 1$. Entry $D[q]$ contains the result of executing the operations with sequence numbers from 0 to $q$, starting from $D[0]$ and replaces the state $s$ stored by the clients in Algorithm 1.

When she receives the INVOKE message of an operation $o$, the server responds by sending pending operations $\omega$ and a response $r$ as before, but includes also authentication data $A$ computed from $\langle d', A', r \rangle \leftarrow \text{authexec}(D[c], A[c], o)$. Note that $d'$ is discarded and that $S$ uses $D[c]$ to compute the result, as the client has only applied the operations with sequence numbers $1, \ldots, c$ at the time when he invokes $o$.

In COP the client checks for commutativity between an invoked operation and the pending operations by himself. Hence, we also require the server to include additional state information that allows the client to execute $\text{commute}_F$. In practice, the server will only store the latest state $D[b]$ and the state changes induced by all operations with sequence numbers $q < b$. Moreover, once $S$ learns from INVOKE messages that all clients have received and applied all operations with a sequence number $q$, then she may discard the state changes for $q$ as well.

### 4.3 Client

Clients no longer maintain the complete state to execute operations, as this is done by the server. Instead, a client $C_i$ stores a chain of digests $G$ such that $G[0]$ is the digest of the initial state $D_0$ and $G[q]$ is the digest of the state that results from applying all operations with sequence numbers $1, \ldots, q$ to $D_0$. Client $C_i$ uses $G$ to verify responses to his operations upon receiving REPLY message. In particular, $C_i$ executes Algorithm 1 and computes $\langle a, r' \rangle \leftarrow \text{verify}(G[c], A, u, r)$, where $C_i$ halts if the original algorithm halts or if $r' = \bot$. If verification step passes, the client augments the COMMIT message by adding $A$ and $r'$ and signs the entire message. Note that $a$ is again used only as a temporary digest for verifying the pending-self operations and is discarded when the method returns.

Upon receiving a BROADCAST message, the client verifies the signature from client $C_j$ that invoked the operation and the hash value as before. Then $C_i$ verifies that the response and state update between him and $C_j$ are consistent by executing $\langle a, r' \rangle \leftarrow \text{verify}(G[q], A, o, r)$. If $r' = \bot$, the client halts; otherwise $r' = r$ and the client updates the digest chain to reflect $o$ by computing the next digest value $G[c + 1]$. Note that $C_i$ cannot use $A$ to update the digest, as $A$ authenticates $o$ in the state where $C_j$ committed it, but this state may differ from $C_i$’s current state; hence, $A$ may not be valid for data structure $D[c]$ and $G[c]$. We therefore require the server to send an additional authenticator $A'$ for $o$ in state $D[c]$. The client verifies that $A'$ and $r$ correspond to $o$ by executing $\langle G[c + 1], r' \rangle \leftarrow \text{verify}(G[c], A', o, r)$, and verifying that $r' \neq \bot$. The client may garbage-collect entries in $G$ in a similar way as for the hash chain in Section 3.
5 Conclusion

This paper has introduced the Commutative-Operation verification Protocol (COP), which allows a group of clients to execute a generic service coordinated by a remote untrusted server. COP ensures fork-linearizability and allows clients to easily verify the consistency and integrity of the service responses. In contrast to previous work, COP is wait-free and supports commuting operation sequences, but may sometimes abort conflicting operations.

Given the popularity of outsourced computation and the cloud-computing model, the problem of checking the integrity of remote computations has received a lot of attention recently \cite{7,22,9,2}. But such cryptographic protocols typically address only a two-party model with a single client. Moreover, the verification is usually done for read-only computations — updates and state changes are not supported. On the other hand, authenticated data structures \cite{21} allow to verify integrity of queries for atomic data updates and have been extended to a multi-client model with fork-linearizability properties \cite{3}. Integrating state updates into verifiable computation and COP or other protocols that guarantee fork-linearizability will represent an important step toward a comprehensive consistency-verification solutions for realistic distributed systems.

Acknowledgments

This work has been supported in part by the European Union’s Seventh Framework Programme (FP7/2007–2013) under grant agreement number ICT-257243 T CLOUDS.

References

[1] M. K. Aguilera, S. Frølund, V. Hadzilacos, S. L. Horn, and S. Toueg, “Abortable and query-abortable objects and their efficient implementation,” in Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC), 2007.

[2] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish, “Verifying computations with state,” in Proc. 24th ACM Symposium on Operating Systems Principles (SOSP), pp. 341–357, 2013.

[3] C. Cachin, “Integrity and consistency for untrusted services,” in Proc. 37th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011), vol. 6543 of Lecture Notes in Computer Science, pp. 1–14, Springer, 2011.

[4] C. Cachin, I. Keidar, and A. Shraer, “Fork sequential consistency is blocking,” Information Processing Letters, vol. 109, pp. 360–364, Mar. 2009.

[5] C. Cachin, I. Keidar, and A. Shraer, “Fail-aware untrusted storage,” SIAM Journal on Computing, vol. 40, pp. 493–533, Apr. 2011. Preliminary version appears in Proc. DSN 2009.

[6] C. Cachin, A. Shelat, and A. Shraer, “Efficient fork-linearizable access to untrusted shared memory,” in Proc. 26th ACM Symposium on Principles of Distributed Computing (PODC), pp. 129–138, 2007.

[7] G. Cormode, M. Mitzenmacher, and J. Thaler, “Practical verified computation with streaming interactive proofs,” in Proc. 3rd Conference on Innovations in Theoretical Computer Science (ITICS), pp. 90–112, 2012.
[8] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, “SPORC: Group collaboration using untrusted cloud resources,” in Proc. 9th Symp. Operating Systems Design and Implementation (OSDI), 2010.

[9] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, “Quadratic span programs and succinct NIZKs without PCPs,” in Advances in Cryptology: EUROCRYPT 2013, vol. 7881 of Lecture Notes in Computer Science, Springer, 2013.

[10] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter, “Zzyzx: Scalable fault tolerance through Byzantine locking,” in Proc. 40th International Conference on Dependable Systems and Networks (DSN-DCCS), 2010.

[11] M. Herlihy, “Wait-free synchronization,” ACM Transactions on Programming Languages and Systems, vol. 11, pp. 124–149, Jan. 1991.

[12] M. Herlihy, V. Luchangco, and M. Moir, “Obstruction-free synchronization: Double-ended queues as an example,” in Proc. 23rd Intl. Conference on Distributed Computing Systems (ICDCS), 2003.

[13] M. P. Herlihy and J. M. Wing, “Linearizability: A correctness condition for concurrent objects,” ACM Transactions on Programming Languages and Systems, vol. 12, pp. 463–492, July 1990.

[14] J. Li, M. Krohn, D. Mazieres, and D. Shasha, “Secure untrusted data repository (SUNDR),” in Proc. 6th Symp. Operating Systems Design and Implementation (OSDI), pp. 121–136, 2004.

[15] J. Li and D. Mazières, “Beyond one-third faulty replicas in Byzantine fault-tolerant systems,” in Proc. 4th Symp. Networked Systems Design and Implementation (NSDI), 2007.

[16] N. A. Lynch, Distributed Algorithms. San Francisco: Morgan Kaufmann, 1996.

[17] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, “Depot: Cloud storage with minimal trust,” in Proc. 9th Symp. Operating Systems Design and Implementation (OSDI), 2010.

[18] M. Majuntke, D. Dobre, M. Serafini, and N. Suri, “Abortable fork-linearizable storage,” in Proc. 13th Conference on Principles of Distributed Systems (OPDIS), vol. 5923 of Lecture Notes in Computer Science, pp. 255–269, Springer, 2009.

[19] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine, “A general model for authenticated data structures,” Algorithmica, vol. 39, pp. 21–41, 2004.

[20] D. Mazieres and D. Shasha, “Building secure file systems out of Byzantine storage,” in Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC), 2002.

[21] M. Naor and K. Nissim, “Certificate revocation and certificate update,” IEEE Journal on Selected Areas in Communications, vol. 18, pp. 561–570, Apr. 2000.

[22] S. Setty, V. Yu, N. Panpalia, B. Braun, A. J. Blumberg, , and M. Walfish, “Taking proof-based verified computation a few steps closer to practicality,” in Proc. 21st USENIX Security Symposium, 2012.

[23] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket, “Venus: Verification for untrusted cloud storage,” in Proc. Cloud Computing Security Workshop (CCSW), ACM, 2010.
[24] R. Tamassia, “Authenticated data structures,” in *Proc. 11th European Symposium on Algorithms (ESA)*, vol. 2832 of *Lecture Notes in Computer Science*, pp. 2–5, Springer, 2003.

[25] W. E. Weihl, “Commutativity-based concurrency control for abstract data types,” *IEEE Transactions on Computers*, vol. 37, pp. 1488–1505, Dec. 1988.

[26] G. Weikum and G. Vossen, *Transactional Information Systems*. Morgan Kaufmann, 2002.

[27] P. Williams, R. Sion, and D. Shasha, “The blind stone tablet: Outsourcing durability to untrusted parties,” in *Proc. Network and Distributed Systems Security Symposium (NDSS)*, 2009.
A Formal definitions

A.1 Consistency properties

We use the standard notion of linearizability [13], which requires that the operations of all clients appear to execute atomically in one sequence and its extension to fork-linearizability [20][6], which relaxes the condition of one sequence to permit multiple “forks” of an execution. Under fork-linearizability, every client observes a linearizable history and when an operation is observed by multiple clients, the history of events occurring before the operation is the same.

Definition 1 (View). A sequence of events \( \pi \) is called a view of a history \( \sigma \) at a client \( C_i \) w.r.t. a functionality \( F \) if:

1. \( \pi \) is a sequential permutation of some subsequence of complete operations in \( \sigma \);
2. all complete operations executed by \( C_i \) appear in \( \pi \); and
3. \( \pi \) satisfies the sequential specification of \( F \).

Definition 2 (Linearizability [13]). A history \( \sigma \) is linearizable w.r.t. a functionality \( F \) if there exists a sequence of events \( \pi \) such that:

1. \( \pi \) is a view of \( \sigma \) at all clients w.r.t. \( F \); and
2. \( \pi \) preserves the real-time order of \( \sigma \).

Definition 3 (Fork-linearizability [20]). A history \( \sigma \) is fork-linearizable w.r.t. a functionality \( F \) if for each client \( C_i \) there exists a sequence of events \( \pi_i \) such that:

1. \( \pi_i \) is a view of \( \sigma \) at \( C_i \) w.r.t. \( F \); and
2. \( \pi_i \) preserves real-time order of \( \sigma \); and
3. for every client \( C_j \) and every operation \( o \in \pi_i \cap \pi_j \) it holds that \( \pi_i|_o = \pi_j|_o \).

Finally, we recall the concept of a fork-linearizable Byzantine emulation [6]. It summarizes the requirements put on our protocol, which runs between the clients and an untrusted server. This notion means that when the server is correct, the service should guarantee the standard notion of linearizability; otherwise, it should ensure fork-linearizability.

Definition 4 (Fork-linearizable Byzantine emulation [6]). We say that a protocol \( P \) for a set of clients emulates a functionality \( F \) on a Byzantine server \( S \) with fork-linearizability if and only if in every fair and well-formed execution of \( P \), the sequence of events observed by the clients is fork-linearizable with respect to \( F \), and moreover, if \( S \) is correct, then the execution is linearizable w.r.t. \( F \).

A.2 Cryptographic primitives

As the focus of this work is on concurrency and correctness and not on cryptography, we model hash functions and digital signature schemes as ideal, deterministic functionalities implemented by a distributed oracle.

A hash function maps a bit string of arbitrary length to a short, unique representation. The functionality provides only a single operation hash; its invocation takes a bit string \( x \) as parameter and returns an integer \( h \) with the response. The implementation maintains a list \( L \) of all \( x \) that have been queried so far. When the invocation contains \( x \in L \), then hash responds with the index of \( x \) in \( L \); otherwise, hash appends \( x \) to \( L \) and returns its index. This ideal implementation models only collision resistance but no other properties of real hash functions.
The functionality of the digital signature scheme provides two operations, \( \text{sign}_i \) and \( \text{verify}_i \). The invocation of \( \text{sign}_i \) specifies the index \( i \) of a client and takes a bit string \( m \) as input and returns a signature \( \sigma \in \{0,1\}^* \) with the response. Only \( C_i \) may invoke \( \text{sign}_i \). The operation \( \text{verify}_i \) takes a putative signature \( \sigma \) and a bit string \( m \) as parameters and returns a Boolean value with the response. Its implementation satisfies that \( \text{verify}_j(\sigma, m) \) returns \( \text{TRUE} \) for any \( i \in \{1, \ldots, n\} \) and \( m \in \{0, 1\}^* \) if and only if \( C_i \) has executed \( \text{sign}_i(m) \) and obtained \( \sigma \) before; otherwise, \( \text{verify}_j(\sigma, m) \) returns \( \text{FALSE} \). Every client as well as \( S \) may invoke \( \text{verify} \). The signature scheme may be implemented analogously to the hash function.

B Analysis

**Theorem 1.** The commutative-operation verification protocol in Algorithms 1–3 emulates functionality \( F' \) on a Byzantine server with fork-linearizability.

We prove this theorem through the sequence of the following lemmas. We start by introducing additional notation.

When a client issues a \text{COMMIT} signature for some operation \( o \), we say that he \text{commits} \( o \). The client’s sequence number included in the signature thus becomes the \text{sequence number of} \( o; \) note that with a faulty \( S \), two different operations may be committed with the same sequence number by separate clients.

**Lemma 2.** If the server is correct, then every history \( \sigma \) is linearizable w.r.t. \( F' \). Moreover, if the clients execute all operations sequentially, then \( \sigma \) is linearizable w.r.t. \( F \).

**Proof.** Recall that \( \sigma \) consists of invocation and response events. We construct a sequential permutation \( \pi \) of \( \sigma \) in terms of the operations associated to the events in \( \sigma \). Note that a client sends an \text{INVOKE} message with his operation to the server, the server assigns a sequence number to the operation and sends it back. The client then computes the response and sends a signed \text{COMMIT} message to \( S \), containing the operation and its sequence number. Since each executed operation appears in \( \sigma \) in terms of its invocation and response events, \( \pi \) contains all operations of all clients.

We order \( \pi \) by the sequence number of the operations. If the server is correct she processes \text{INVOKE} messages in the order they are received and assigns sequence numbers accordingly. This implies that if an operation \( o' \) is invoked after an operation \( o \) completes, then the sequence number of \( o' \) is higher than \( o \)’s. Hence, \( \pi \) preserves the real-time order of \( \sigma \).

We now use induction on the operations in \( \pi \) to show that \( \pi \) satisfies the sequential specification of \( F' \). Note that \( F' \) requires a bit of care, as it is not deterministic. For a sequence \( \omega \) of operations of \( F' \) in an actual execution, we write \( \text{successful}(\omega) \) for the subsequence whose status was \text{SUCCESS}; restricted to such operations, \( F' \) is deterministic. In particular, consider some operation \( o \in \pi \), executed by client \( C_i \). We want to show that \( C_i \) computes \( (s', r) \) such that \( (s', r) \in F'(s_0, \text{successful}(\pi[\omega])) \), whereby it outputs \( r \) after committing \( o \) and stores \( s' \) in its variable \( s \) after applying \( o \).

Consider the base case where \( o \) is the first operation in \( \pi \). Note that \( S \) has not reported any pending operations to \( C_i \) because \( o \) is the first operation. Thus, \( C_i \) determines that the status of \( o \) is \text{SUCCESS}, computes \( (s', r) \leftarrow F(s_0, o) \) and outputs \( r \). Hence, \( F' \) is satisfied. When \( C_i \) later receives \( o \) in the \text{BROADCAST} message from \( S \) with sequence number \( 1 \), the state is also updated correctly.

Now consider the case when \( o \) is not the first operation in \( \pi \) and assume that the induction assumption holds for an operation that appears in \( \pi \) before \( o \). If the status of \( o \) is \text{ABORT}, then the client does not invoke \( F \), returns \( \perp \), and leaves the state unchanged upon applying \( o \). The claim follows.

Otherwise, we need to show that the response \( r \neq \perp \) and the state \( s' \) after applying \( o \) satisfy \( (s', r) = F(s_0, \text{successful}(\pi[\omega])) \). Since \( S \) is correct, she assigns unique sequence numbers to the operations.
We split the operations with a sequence number smaller than that of \( o \) in three groups: a sequence \( \rho \) of operations that \( C_i \) has confirmed before he committed \( o \), this sequence is in the order in which \( C_i \) confirmed these operations; a sequence \( \delta \) of operations of other clients that were reported by \( S \) as pending to \( C_i \) when executing \( o \), ordered as in the \textsc{Reply} message; and a sequence \( \nu \) of operations that \( C_i \) has committed itself before \( o \) but not yet confirmed or applied, ordered by their sequence number.

Observe that \( C_i \) computes \( r \) starting from its own copy of the state \( s \) that results after applying all operations in \textsc{successful}(\( \rho \)). From the induction assumption, it follows that \( (\hat{s}, \cdot) = F(s_0, \textsc{successful}(\rho)) \) because \( \rho \) is a prefix of \( \pi \). From variable \( \omega \) in the \textsc{Reply} message, \( C_i \) computes the pending-other operations \( \gamma \) and the successful pending-self operations \( \mu \). Note that \( \gamma = \delta \) and \( \mu = \textsc{successful}(\nu) \) as the server is correct. The client computes a temporary state \( (a, \cdot) = F(\hat{s}, \mu) \). Because \( o \) does not abort, \( C_i \) has determined that \( o \) commutes with \( \gamma \) in \( a \) and computed \( \cdot, r) = F(a, o) \). By the definition of commuting operation sequences, we have that \( (s', r) = F(a, \textsc{successful}(\gamma) \circ o) \) and \( (s', r) = F(\hat{s}, \textsc{successful}(\omega)) \) since the order of operations in \( \mu \) and \( \gamma \) is preserved in \( \omega \). Hence, \( (s', r) = F(s_0, \textsc{successful}(\pi | o)) \).

The sequence \( \pi \) preserves the real-time order of \( \sigma \) and satisfies the three conditions of a view of \( \sigma \) at every client \( C_i \) w.r.t. \( F \), hence, \( \sigma \) is linearizable w.r.t. \( F' \).

The second part of the lemma claims that if clients execute operations sequentially, then no client outputs \( \bot \). Since the sequence of events at every client is well-formed, a client does not invoke an operation before it has committed the previous one. Moreover, if clients execute operations sequentially then no client invokes an operation while there is a client who has not completed his operation. Hence, the server never includes any pending operations in \( \omega \) of the \textsc{Reply} message. The check for conflicts is never positive, and all operations have status \textsc{success}. Hence, no client returns \( \bot \) and \( \sigma \) satisfies the sequential specification of \( F \).

\textbf{The promised view of an operation.} Suppose a client \( C_i \) executes and thereby commits an operation \( o \). We define the \textit{promised view} to \( C_i \) of \( o \) as the sequence of all operations that \( C_i \) has confirmed before committing \( o \), concatenated with the sequence \( \omega \) of pending operations received in the \textsc{Reply} message during the execution of \( o \), including \( o \) itself (according to the protocol \( C_i \) verifies that the last operation in \( \omega \) is \( o \)).

\textbf{Lemma 3.} If \( C_j \) has confirmed some operation \( o \) that was committed by a client \( C_i \), then the sequence of operations that \( C_j \) has confirmed up to (and including) \( o \) is equal to the promised view to \( C_i \) of \( o \) In particular,

1. if \( C_i \) and \( C_j \) have confirmed an operation \( o \), then they have both confirmed the same sequence of operations up to \( o \); and
2. the promised view to \( C_i \) of \( o \) contains all operations executed by \( C_i \) up to \( o \).

\textbf{Proof.} Note that every client computes a hash chain \( H \) in which every defined entry contains a hash value that represents a sequence of operations. More precisely, if \( C_i \) commits \( o \) with sequence number \( l \), then he has set \( H[l] \leftarrow \text{hash}(H[l-1]||o||l); \) this step recursively defines the sequence represented by \( H[l] \) as the sequence represented by \( H[l-1] \) followed by \( o \). According to the collision-resistance of the hash function, no two different operation sequences are represented by the same hash value. Note that no client ever overwrites an entry of \( H \); moreover, if a client arrives at a point in the protocol where he might assign some value \( h \) to entry \( H[l] \) but \( H[l] \neq \bot \), then he verifies that \( H[l] = h \) and aborts if this fails.

Consider the moment when \( C_i \) receives the \textsc{Reply} message during the execution of \( o \). The view of \( o \) promised to \( C_i \) contains the sequence of operations that \( C_i \) has confirmed, followed by the list \( \omega \) in the \textsc{Reply} message, including \( o \).
For every pending operation \( p \in \omega \), client \( C_i \) checks if he has already an entry in \( H \) at index \( l \), which is the promised sequence number of \( p \) to \( C_i \) according to \( \omega \). If there is no such entry, he computes the hash value \( H[l] \) as above. Otherwise, \( C_i \) must have received an operation for sequence number \( l \) earlier, and so he verifies that \( o \) is the same pending operation as received before. Moreover, \( C_i \) verifies that his last invoked operation is also returned to him as pending and adds it to \( H \). Hence, the new hash value \( h \) stored in \( H \) at the sequence number of \( o \) represents the promised view to \( C_i \) of \( o \).

Subsequently, \( C_i \) signs \( o \) and \( h \) together and sends it to the server. Client \( C_j \) receives it in a broadcast message from \( S \), to be confirmed and applied with sequence number \( q \). Because \( C_j \) verifies the signature of \( C_i \) on \( o \), \( q \), and \( h \), the hash value \( h \) received by \( C_j \) represents the promised view to \( C_i \) of \( o \). Before \( C_j \) applies \( o \) as his \( q \)-th operation, according to the protocol he must have already confirmed \( q - 1 \) operations one by one. Client \( C_j \) also verifies that he has either already computed the same \( H[q] = h \) or he computes \( H[q] \) from his value \( H[q - 1] \) and checks \( H[q] = h \). As \( H[q] \) represents the sequence of operations that \( C_j \) has confirmed up to \( o \), from the collision resistance of the hash function, this establishes the main statement of the lemma.

The first additional claim follows simply by noticing that the statement of the lemma holds for \( i = j \). For showing the second additional claim, we note that if \( C_i \) confirms an operation of himself, then he has previously executed it (successful or not). There may be additional operations that \( C_i \) has executed but not yet confirmed, but \( C_i \) has verified according to the above argument that these were all contained in \( \omega \) from the REPLY message. Thus they are also in the promised view of \( o \).

The view of a client. We construct a sequence \( \pi_i \) from \( \sigma \) as follows. Let \( o \) be the operation committed by \( C_i \) which has the highest sequence number among those operations of \( C_i \) that have been confirmed by some client \( C_k \) (including \( C_i \)). Define \( \alpha_i \) to be the sequence of operations confirmed by \( C_k \) up to and including \( o \). Furthermore, let \( \beta_i \) be the sequence of operations committed by \( C_i \) with a sequence number higher than that of \( o \). Then \( \pi_i \) is the concatenation of \( \alpha_i \) and \( \beta_i \). Observe that by definition, no client has confirmed operations from \( \beta_i \).

Lemma 4. The sequence \( \pi_i \) is a view of \( \sigma \) at \( C_i \) w.r.t. \( F' \).

Proof. Note that \( \pi_i \) is defined through a sequence of operations that are contained in \( \sigma \). Hence \( \pi_i \) is sequential by construction.

We now argue that all operations executed by \( C_i \) are included in \( \pi_i \). Recall that \( \pi_i = \alpha_i \circ \beta_i \) and consider \( o \), the last operation in \( \alpha_i \). As \( o \) has been confirmed by \( C_k \), Lemma 3 shows that \( \alpha_i \) is equal to the promised view to \( C_i \) of \( o \) and, furthermore, that it contains all operations that \( C_i \) has executed up to \( o \). By construction of \( \pi_i \) all other operations executed by \( C_i \) are contained in \( \beta_i \), and the property follows.

The last property of a view requires that \( \pi_i \) satisfies the sequential specification of \( F' \). Note that \( F' \) is not deterministic and some responses might be \( \bot \). But when we ensure that two operation sequences of \( F' \) have responses equal to \( \bot \) in exactly the same positions, then we can conclude that two equal operation sequences give the same resulting state and responses from the fact that \( F \) is deterministic.

We first address the operations in \( \alpha_i \). Consider again \( o \), the last operation in \( \alpha_i \), which has been confirmed by \( C_k \). For the point in time when \( C_i \) executes \( o \), define \( \rho \) to be the sequence of operations that \( C_i \) has confirmed prior to this and define \( s \) as the resulting state from applying the successful operations in \( \rho \), as stored in variable \( s \); furthermore, let \( \omega \) be the pending operations contained in the REPLY message from \( S \). Observe that \( \omega \) can be partitioned in the pending-other operations \( \gamma \), the successful pending-self operations of \( C_i \) as stored in \( \mu \), the aborted pending-self operations of \( C_i \), and \( o \). Client \( C_i \) computes the response \( r \) for \( o \) in state \( a \) that results from \( F(s, \mu) \). Before executing \( o \), \( C_i \) verifies that \( o \) commutes with \( \gamma \) in \( a \). Note that when \( C_i \) committed some operation \( p \in \mu \) he has also verified that \( p \)
commuted with the pending-other operations in $\omega|\rho$. Hence, the response resulting from executing the operations in $\rho$ followed by $\mu \circ o$ is the same as that resulting from executing $\mu \circ \text{successful}(\gamma) \circ o$ after $\rho$ (recall the notation $\text{successful}(\cdot)$ from Lemma 2). Since $\omega$ preserves the order of operations in $\mu$ and $\gamma$, the response is also the same after the execution of $\rho \circ \text{successful}(\omega)$. Moreover, the state resulting from executing the operations in $\rho$ followed by $\mu \circ \text{successful}(\gamma) \circ o$ is the same as that resulting from executing $\rho \circ \text{successful}(\omega)$. Since $\rho \circ o$ is the promised view to $C_i$ of $o$, and since $C_k$ has confirmed $o$, Lemma 3 now implies that $\rho \circ o$ is equal to $\alpha_i$.

To conclude the argument, we only have to show that the abort status for all operations in the sequences is the same. Then they will produce the same responses and the same final state. Note that when $C_i$ executes some operation $o$ he either computes a response according to $F$ or aborts the operation, declaring its status to be SUCCESS or ABORT, respectively. For operations in $\rho$ this is clear from the protocol as the status is included in the BROADCAST message. And whenever $C_i$ later obtains $o$ again as a pending-self operation in $\omega$ at some index $l$, he verifies that it is the same operation as previously at index $l$ and applies or skips it as before according to the status remembered in $Z[l]$. Hence, the responses of $C_i$ from executing the operations in $\alpha_i$ respect the specification of $F'$. The remainder of $\pi_i$ consists of $\beta_i$, whose operations $C_i$ executes himself using $F'$. Hence, $\pi_i$ satisfies the sequential specification of $F'$.

**Lemma 5.** If some client $C_k$ confirms an operation $o_1$ before an operation $o_2$, then $o_2$ does not precede $o_1$ in the execution history $\sigma$.

**Proof.** Let $\delta_k$ denote the sequence of operations that $C_k$ has confirmed up to $o_2$. According to the protocol logic, $\delta_k$ contains $o_1$, and $o_1$ has a smaller sequence number than $o_2$. Lemma 5 shows that $\delta_k$ is equal to the promised view to $C_k$ of $o_2$, hence, $o_1$ is in the promised view to $C_k$ of $o_2$. Recall that the promised view contains operations that have been committed or are pending for other clients. Hence, $o_1$ has been invoked before $o_2$ completed.

**Lemma 6.** The sequence $\pi_i$ preserves the real-time order of $\sigma$.

**Proof.** Recall that $\pi_i = \alpha_i \circ \beta_i$ and consider first those operations of $\pi_i$ that appear in $\alpha_i$, that is, they have been confirmed by some client $C_k$. Lemma 5 shows that these operations preserve the real-time order of $\sigma$. Second, the operations in $\beta_i$ are ordered according to their sequence number and they were committed by $C_i$. According to the protocol, $C_i$ executes only one operation at a time and always assigns a sequence number that is higher than the previous one. Hence, $\beta_i$ also preserves the real-time order of $\sigma$.

We are left to show that no operation in $\beta_i$ precedes an operation from $\alpha_i$ in $\sigma$. Recall that $\alpha_i$ is the promised view to $C_i$ of $o$ (the last operation in $\alpha_i$) and includes the operations that $C_i$ has confirmed or received as pending from $S$ after $C_i$ invoked $o$. Since $o$ precedes all operations from $\beta_i$, it follows that no operation in $\alpha_i$ precedes an operation from $\beta_i$.

**Lemma 7.** If $o \in \pi_i \cap \pi_j$ then $\pi_i|_o = \pi_j|_o$.

**Proof.** As $\pi_i = \alpha_i \circ \beta_i$ and $\pi_j = \alpha_j \circ \beta_j$, we need to consider four cases to analyze all operations that can appear in $\pi_i \cap \pi_j$ and the rest are symmetrical.

1. $o \in \alpha_i$ and $o \in \alpha_j$: This case happens when (a) $C_i$ and $C_j$ both confirmed $o$, or when (b) $C_i$ has confirmed an operation of $C_j$ or vice versa, or when (c) a client $C_k$ has confirmed operations of $C_i$ and $C_j$. For (a) and (b) Lemma 3 shows that $\alpha_i|_o = \alpha_j|_o$. In case (c) neither $C_i$ nor $C_j$ has confirmed $o$, but $o$ is in their views because $C_k$ has confirmed pending operations of $C_i$ and $C_j$. Hence, $\pi_k|_o = \alpha_i|_o$ and $\pi_k|_o = \alpha_j|_o$ again from Lemma 3.
2. \( o \in \beta_i \) and \( o \in \alpha_j \): This case cannot happen, since no client has confirmed operations from \( \beta_i \) by definition.

3. \( o \in \alpha_i \) and \( o \in \beta_j \): Analogous to the case above.

4. \( o \in \beta_i \) and \( o \in \beta_j \): This case cannot happen since \( \beta_i \) and \( \beta_j \) contain only pending-self operations of \( C_i \) and \( C_j \), correspondingly.