Verifying the Consistency of Remote Untrusted Services with Commutative Operations

Christian Cachin[1] and Olga Ohrimenko[2,*]

[1] IBM Research - Zurich, Switzerland
cca@zurich.ibm.com
[2] Microsoft Research, Cambridge, United Kingdom
oohrim@microsoft.com

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 first presents the *Commutative-Operation verification Protocol (COP)* that ensures linearizability when the server is correct and preserves fork-linearizability in any other case. 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. Second, this work extends COP through authenticated data structures to *Authenticated COP* , which allows consistency verification of outsourced services whose state is kept only remotely, by the server. This yields the first fork-linearizable consistency verification protocol for generic outsourced services that (1) *relieves* clients from *storing the state*, (2) supports *wait-free* client operations, and (3) handles *sequences* of arbitrary *commutative operations*.

Keywords: cloud computing, fork-linearizability, data integrity, verifiable computation, commutative operations, Byzantine emulation.

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.

[*] Work done at IBM Research - Zurich and at Brown University.

M.K. Aguilera et al. (Eds.): OPODIS 2014, LNCS 8878, pp. 1–16, 2014.

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 forked clients cannot detect such violations. The best achievable consistency guarantee in this setting is captured by *fork-linearizability*, introduced by Mazières and Shasha [23] for storage systems. Fork-linearizability ensures that whenever the server in her responses to a client C_1 has ignored an operation executed by a client C_2, then C_1 can never again observe an operation by C_2 afterwards and vice versa. This property ensures clearly defined service semantics in the face of an attack and allows clients to detect server misbehavior easily.

Several conceptual [8, 21, 5, 6] and practical advances [29, 13, 20, 27] have recently been made that improve consistency checking and verification with fork-linearizability and related notions. 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 cause such protocols to block; this applies to fork-linearizable semantics [23, 8] and to other forking consistency notions [5, 6].

In this paper, we go beyond storage services and verify the consistency of remote *computation* on a Byzantine server. The *Commutative-Operation verification Protocol* or *COP* imposes fork-linearizable semantics for arbitrary functionalities, exploits commuting operations, and allows clients to operate concurrently without blocking unless operations conflict. Furthermore, the extension to *Authenticated COP* also relieves clients from storing the computation state and from executing all operations. Fork-linearizability makes it easy to expose Byzantine behavior of the server. For instance, the clients may exchange a message outside the model over a low-bandwidth channel and thereby verify the correctness of a service in an end-to-end way.

Efficient handling of wait-free operations is a key feature for collaboration with remote coordination, as geographically separated clients may operate at different speed. Consequently, previous work has devoted a lot of attention to identifying and avoiding blocking [23, 8, 19]. For example, read operations in a storage service commute and do not lead to a conflict. On the other hand, when a client writes a data item concurrently with another client who reads it, the reader has to wait until the write operation completes; otherwise, fork-linearizability is not guaranteed [8]. If all operations are to proceed without blocking, though, it is necessary to weaken the consistency guarantees to weak fork-linearizability [6], for instance. COP is wait-free and never blocks because it aborts non-commuting operations that cannot proceed.

The *Blind Stone Tablet (BST)* protocol [29], 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. Every client has to maintain the complete service state and to execute all operations, in contrast to this work. Furthermore, the BST protocol guarantees fork-linearizability only for database state updates, but does not ensure it for all responses output by a client.

SPORC [13] 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 but respect only fork-* linearizability, which is weaker than fork-linearizability.

Contributions. This paper considers a generic service executed by an untrusted server and provides new protocols for consistency verification through fork-linearizable semantics. More concretely, it introduces the Commutative-Operation verification Protocol (COP) and its extension to Authenticated COP (called ACOP) with the following properties:

1. COP is the first wait-free, abortable consistency verification protocol that emulates an arbitrary functionality on a Byzantine server with fork-linearizability and exploits commuting operation sequences. (See Sect. 3.)
2. ACOP is the first wait-free fork-linearizable consistency verification protocol for services, where the state is maintained by the server and the clients do not execute every operation. (See Sect. 4.)
3. COP comes with a formal analysis that proves fork-linearizable semantics for generic service execution; previous work did not establish this notion.

COP and ACOP follow the general pattern of most previous fork-linearizable emulation protocols. For determining when to proceed with concurrent operations, we consider *sequences* of operations that jointly commute and the state of the service, in contrast to earlier protocols, which considered only isolated operations. For lack of space, some definitions and the formal analysis are contained in the full version [7].

For computations supported by suitable authenticated data structures, ACOP enables *authenticated remote computation*, where operations are executed by the server and the clients no longer need to maintain the state of the computation. In contrast to previous work, this enables ACOP to handle services with large state.

1.1 Related Work

Storage protocols. Fork-linearizability has been introduced (under the name of *fork consistency*) together with the SUNDR storage system [23, 18]. Conceptually SUNDR operates on storage objects with simple read/write semantics. Subsequent work of Cachin et al. [8] 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. [21].

FAUST [6] and Venus [27] 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 [23]. The relation of these protocols and others to COP is summarized in Tab. 1.

Table 1. Summary of related protocols. In this table under *function*, the BST protocol supports only a *single* commuting operation and does not achieve wait-freedom (as indicated by the parentheses in the first column); SPORC is wait-free for generic functions that have *operational transforms*; COP and ACOP are wait-free for generic *commuting* operation sequences. *Weak fork-linearizability* (or *fork-* consistency*) allows the last operation of a client to be inconsistent compared to *fork-linearizability*; however, BST and SPORC do not guarantee their consistency notion for client responses, only for state changes that may occur much later (as indicated by the parentheses). The *execution* column indicates whether the *clients* compute operations and maintain state or whether this is done by the *server*.

Protocol	Wait-free	Function	Consistency	Execution
SUNDR [23, 18]	—	storage	fork-lin.	server
FAUST & Venus [6, 27]	✓	storage	weak fork-lin.	server
BST [29]	(✓)	single comm. op.	(fork-lin.)	clients
SPORC [13]	✓	generic o.-t. op.	(weak fork-lin.)	clients
COP (Sec. 3)	✓	generic comm. op.	fork-lin.	clients
ACOP (Sec. 4)	✓	generic comm. op.	fork-lin.	server

Blind Stone Tablet (BST). The BST protocol [29] considers transactions on a database, coordinated by the remote server. A client first *simulates* a transaction on its own copy, potentially generating *local output*, then coordinates with the server for ordering the transaction. From the server's response it determines if a 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 several 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 produce the correct output. This implies the client is blocked and the protocol is not "wait-free" as claimed [29]. Second, the notion of "trace consistency" in the analysis of BST considers only transactions that have been applied to the local state, not the responses as required to satisfy fork-linearizability. However, a transaction may be applied long after its response was output, hence, client operations might not be fork-linearizable. In contrast, the analysis of COP shows it is fork-linearizable for all *responses* output by clients. Finally, every client in BST maintains a copy of the database and replays all operations locally, which is not necessary in ACOP.

COP extends BST and 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, considering the current service *state*. 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. Operation sequences and state-based commutativity have recently been exploited for building scalable services on multicore systems [10].

Non-blocking protocols. SPORC [13] 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 is claimed to provide fork-* linearizability [19], which is almost the same as weak fork-linearizability [6]; both notions are strict 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 two clients. Moreover, there is no formal analysis for SPORC. As in BST, SPORC addresses only the updates of client states and does *not* consider *local outputs*; however, for showing linearizability, one has to consider the respones of operations.

FAUST [6], mentioned before, never blocks clients and enjoys eventual consistency, but guarantees only weak fork-linearizability. Abortable operations have been introduced in this context by Majuntke et al. [21] for data storage.

In contrast to SPORC and FAUST, COP ensures the stronger fork-linearizability condition, where every operation is consistent as soon as it completes. In terms of expressiveness, 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, SPORC is limited to functions with transformable operations and does not address conflicting operations (which exist in some functions [8]); COP, however, works for arbitrary functions.

In BST and SPORC, all clients execute all operations. ACOP eliminates this drawback and shifts the state and the computation to the server by exploiting the notion of authenticated data structures, as suggested by Cachin [3] in a more restricted setting. In storage protocols (SUNDR and FAUST), clients do not "execute" each other's operations due to the limited functionality.

Last but not least, the protocol of Cachin [3] provides also fork-linearizable execution for generic services like COP. However, the protocol is inherently blocking.

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* $\mathcal{S}$ and a set of *operations* $\mathcal{O}$. F takes as arguments a state $s \in \mathcal{S}$ and an operation $o \in \mathcal{O}$ and returns a tuple (s', r), where $s' \in \mathcal{S}$ is a state that reflects any changes that o caused to s and $r \in \mathcal{R}$ is a response to o i.e., $(s', r) = F(s, o)$. 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 $(s', r) = (s_k, r_k)$. Note that an operation in $\mathcal{O}$ 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 $o_1, o_2 \in \mathcal{O}$ 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 $(s', r_1) \leftarrow F(s, o_1)$, $(s'', r_2) \leftarrow F(s', o_2)$; and $(t', q_2) \leftarrow F(s, o_2)$, $(t'', q_1) \leftarrow F(t', o_1)$, then $r_1 = q_1$, $r_2 = q_2$, and $s'' = t''$. Furthermore, we say two operations $o_1, o_2 \in \mathcal{O}$ *commute* when they commute in any state of $\mathcal{S}$.

Also sequences of operations can commute. Suppose two sequences ρ_1 and ρ_2 consisting of operations in $\mathcal{O}$ are mixed together into one sequence π such that the partial order among the operations from ρ_1 and from ρ_2 is retained in π, respectively. If executing π 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 ρ_1 and ρ_2 *commute in state* s. Analogously, we say that ρ_1 and ρ_2 *commute* if they commute in any state.

Operations that do not commute are said to *conflict*. We define a Boolean predicate $\mathit{commute}_F(s, \rho_1, \rho_2)$ that is true if and only if ρ_1 and ρ_2 commute in s according to F. W.l.o.g. we assume all operations of F and $\mathit{commute}_F$ are efficiently computable.

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, 21], we allow operations to abort and augment F to an *abortable* functionality F' accordingly. F' is defined over the same set of states $\mathcal{S}$ and operations $\mathcal{O}$ as F, but returns a tuple defined over $\mathcal{S}$ and $\mathcal{R} \cup \{\perp\}$. 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 $F'(s, o) = \big\{(s, \perp),\ F(s, o)\big\}$. Since F' is not deterministic, a sequence of operations no longer uniquely determines the resulting state and response value.

Abortable functionalities may be seen as obstruction-free objects [1, 15] and vice versa; such objects guarantee that every client operation completes assuming the client eventually runs in isolation.

Operations, histories, and consistency properties. Clients interact with F via operations. 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 response events. We use the standard notions of events, precedence, and histories.

The condition of *linearizability* [16] requires that the operations of all clients appear to execute atomically in one sequence, and its extension to *fork-linearizability* [23, 8],

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* [8] 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 the full version [7].

Cryptography. We make use of two cryptographic primitives, namely a collision-free hash function *hash* and a digital signature scheme, with operations denoted by $sign_i$ and $verify_i$ 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 (see [4]).

3 The Commutative-Operation Verification Protocol

Notation. The function $length(a)$ for a list a denotes the number of elements in a and $\|$ denotes concatenation of strings. 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 $\perp$. Recall that F' is the abortable extension of functionality F.

Overview. COP, presented in Algorithms 1–3, adopts the structure of previous protocols that guarantee fork-linearizable semantics [23, 29, 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 a correct S. (A Byzantine S may introduce consistency violations here.) 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. If this or any other verification step fails, the formal protocol simply halts; in practice, the clients would then recover the service state, abandon the faulty S, and switch to another provider. In order to ensure fork-linearizability for the 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 $r = \perp$. According to this, the *status* of o is a value in $\mathcal{Z} = \{\text{SUCCESS}, \text{ABORT}\}$. 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 broadcast 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 $s \in \mathcal{S}$ 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 $\mathit{hash}(H[l-1] \| o \| l \| i)$, 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 $\perp$. 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 (correct) 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 sequence number. Variable t determines the global sequence number for the invoked operations. Finally, variable b is the sequence number of the last broadcast 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 τ, a digital signature computed over o and i. In turn, a correct S sends a REPLY message with the list ω of pending operations; they have a sequence number greater than c. Upon receiving a REPLY message, the client checks that ω is consistent with any previously sent operations and uses ω to assemble the successful pending-self operations μ and the pending-other operations γ. He then determines whether o can be executed or has to be aborted.

In particular, during the loop in Algorithm 1, for every operation o in ω, C_i determines its sequence number l and verifies from the digital signature that o was indeed invoked by C_j. He computes the entry of o in the hash chain from o, l, j, and $H[l-1]$. If $H[l] = \perp$, then C_i stores the hash value there. Otherwise, $H[l]$ has already been set and 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 μ. The client remembers the status of his own operations in Z, since $\mathit{commute}_F$ depends on the state and that could have changed if he applied operations after committing o.

Finally, when C_i reaches the end of ω (i.e., when C_i considers $o = u$), he checks that ω is not empty and that it contains u at the last position. He then creates a temporary state a by applying μ to the current state s, and tests whether u commutes with the pending-other operations γ 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 γ, 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.

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 $H[0] = \text{NULL}$
$Z : \mathbb{N}_0 \to \mathcal{Z}$: status map (see text), initially empty
$s \in \mathcal{S}$: current state, after applying operations, initially s_0

upon invocation o **do**
  $u \leftarrow o$
  $\tau \leftarrow \mathit{sign}_i(\text{INVOKE}\|o\|i)$
  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 $k \leq \mathit{length}(\omega)$ **do**
    $(o, j, \tau) \leftarrow \omega[k]$
    $l \leftarrow c + k$ // promised sequence number of o
    if not $\mathit{verify}_j(\tau, \text{INVOKE}\|o\|j)$ **then**
      halt
    if $H[l] = \bot$ **then**
      $H[l] \leftarrow \mathit{hash}(H[l-1]\|o\|l\|j)$ // extend hash chain
    else if $H[l] \neq \mathit{hash}(H[l-1]\|o\|l\|j)$ **then** // server replies are inconsistent
      halt
    if $j = i \wedge Z[l] = \text{SUCCESS} \wedge k < \mathit{length}(\omega)$ **then**
      $\mu \leftarrow \mu \circ \langle o \rangle$
    else if $j \neq i$ **then**
      $\gamma \leftarrow \gamma \circ \langle o \rangle$
    $k \leftarrow k + 1$
  if $k = 1 \vee o \neq u \vee j \neq i$ **then** // variables o, j, and $l = c + \mathit{length}(\omega)$ keep their values
    halt // last pending operation must equal the current operation
  $(a, r) \leftarrow F(s, \mu)$ // compute temporary state with successful pending-self operations
  if $\mathit{commute}_F(a, \langle u \rangle, \gamma)$ **then** // $u = o$ is the current operation
    $(a, r) \leftarrow F(a, u)$ // compute response to u
    $Z[l] \leftarrow \text{SUCCESS}$
  else
    $r \leftarrow \bot$
    $Z[l] \leftarrow \text{ABORT}$
  $\phi \leftarrow \mathit{sign}_i\big(\text{COMMIT}\|u\|l\|H[l]\|Z[l]\big)$
  send message $[\text{COMMIT}, u, l, H[l], Z[l], \phi]$ 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 $\big(q = c + 1$ **and** $\mathit{verify}_j(\phi, \text{COMMIT}\|o\|q\|h\|z)\big)$ **then** // server replies are not consistent
    halt
  if $H[q] = \perp$ **then** // operation has not been pending at client
    $H[q] \leftarrow \mathit{hash}(H[q-1]\|o\|q\|j)$
  if $h \neq H[q]$ **then**
    halt // server replies are not consistent
  if $z = \text{SUCCESS}$ **then** // at this point, the operation is confirmed
    $(s, r) \leftarrow F(s, o)$ // 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 broadcast operation, initially 0
  $I : \mathbb{N} \to \mathcal{O} \times \mathbb{N}_0 \times \{0,1\}^*$: invoked operations (see text), initially empty
  $O : \mathbb{N} \to \mathcal{O} \times \{0,1\}^* \times \mathcal{Z} \times \{0,1\}^* \times \mathbb{N}$: committed operations (see text), initially empty

upon receiving message $[\text{INVOKE}, o, c, \tau]$ from C_i **do**
  $t \leftarrow t + 1$
  $I[t] \leftarrow (o, i, \tau)$
  $\omega \leftarrow \langle I[b+1], \ldots, I[t] \rangle$ // 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 $O[b+1] \neq \perp$ **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

Upon receiving a COMMIT message for an operation o with 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 broadcast them. This ensures that completed operations are disseminated in the global order to all clients. Waiting here leads to blocking in BST, as mentioned in the Introduction. In COP, 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, ϕ, j). The client conducts several verification steps; if successful, we say o is *confirmed*. Subsequently he *applies* o to his state s. In more

detail, the client first verifies that the sequence number q is the next operation according to c; hence, o follows the global order and the server did not omit any operations. Second, he uses the digital signature ϕ on the message to verify that C_j indeed committed o. Lastly, C_i 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 *add*(x) operation adds x to s and returns TRUE; a *dec*(x) operation subtracts x from s and returns TRUE if $x \leq s$, but does nothing and returns FALSE if $x > s$. Suppose the current state s at C_i is 7 and C_i executes *dec*(4) and subsequently *dec*(6). During both operations of C_i, the server announces that *add*(2) by another client is pending. Note that C_i executes *dec*(4) successfully but aborts *dec*(6) because *dec*(6) does not commute with *add*(2) from 3, the temporary state (a in Algorithm 1) computed by C_i after the pending-self operation. However, the latter two operations, *add*(2) and *dec*(6), do commute in the current state 7. This shows why the client executes the pending-self operations before testing the current operation for a conflict.

Suppose now the current state s is again 7 and C_i executes *dec*(4). The server reports the pending sequence $\langle$*dec*(2), *dec*$(3)\rangle$. Thus, C_i aborts *dec*(4). Even though *dec*(4) commutes with *dec*(2) and with *dec*(3) individually in state 7, it does not commute with their sequence. This illustrates why COP checks 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 ω. If even one client is slow, then the length of ω 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 $\perp$, 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 [8]. COP avoids blocking via the augmented functionality F'. Clients complete 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. Such synchronization is common for services with strong 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 [18, 27]; these methods apply also to COP.

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; the details appear in the full version [7].

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

In this section, we introduce *Authenticated COP* or *ACOP* , which shifts state maintenance and service execution to the server and lets clients only perform verification. ACOP extends COP with an authenticated data structure [24] for the service functionality. It enables *authenticated remote computation* for many realistic services with complex interfaces[12, 25, 9, 17], such as indexed databases, search trees, document processing services, and generic storage schemes; typically their operations permit queries and updates. Recent advances in cryptographic tools for verifying remote computation suggest that it may even become feasible to construct authenticators for generic computations while preserving the privacy of the inputs [14, 2].

4.1 Authenticated COP

We consider a server that stores shared state and executes operations of the functionality F invoked by clients. When F supports an *authenticated data structure* [24], the clients may verify the integrity of a response to an operation from a cryptographic proof in the form of an authenticator for the response. ACOP results from integrating the authenticated data structure into COP and ensures the fork-linearizability of the service, retaining all other benefits of COP.

More formally, suppose S maintains the state of F in variable x, called the *server's state*; when S receives an operation o from a client, she should update the state by executing $(x', r) \leftarrow F(x, o)$ and send the response r to the client. For adding authentication, the server's state is extended to include authentication data, and an authenticator α is computed with the response as $(x', \alpha, r) \leftarrow \mathit{authexec}_F(x, o)$. The server sends r together with α to the client. The client maintains a *digest* d between operations, which authenticates the (potentially large) state of F maintained by S. For checking the correctness of the response, the client computes $(d', r') \leftarrow \mathit{verify}_F(d, \alpha, o, r)$, whereby $r' = \perp$ indicates that the verification failed, and otherwise, $r' = r$ is the correct response. The $\mathit{authexec}_F$ and verify_F operations encapsulate the authenticated data structure; more information can be found in the rich literature on the subject [28, 22]. For practical authentication techniques such as hash trees and authenticated dictionaries, α is usually much smaller than the full state.

We now describe how to extend Algorithms 1–3 for ACOP.

4.2 Server

We start with the changes for S. As part of her state, S additionally maintains a state map $X : \mathbb{N}_0 \to \{0,1\}^*$ indexed by operations, where $X[0] = s_0$ is the initial state. Entry $X[b]$ is assigned when the server broadcasts an operation with sequence number b such that $X[b]$ contains the result of executing the operations with sequence numbers from $1, \ldots, b$.

When the server receives the INVOKE message from C_i with an operation o, she increments the index t and considers the pending operations ω with index between b and t. Then S executes the pending-self operations ν of C_i, which include o, to obtain the response and authenticator for o as $(x', \alpha, r) \leftarrow \mathit{authexec}_F(X[b], \nu)$; she sends ω

and r to C_i together with α. Note that x' is discarded and that S uses $X[b]$ to compute the result using the operation sequence ν, which includes o, as C_i has only applied the operations with sequence numbers $1, \ldots, b$ at the time when he invokes o.

In COP the client checks for commutativity between an invoked operation and the pending operations by himself. With the above modification, S also needs to abort operations as the client would determine from $\mathit{commute}_F$ when computing r and α, and S must include additional information that allows the client to execute $\mathit{commute}_F$. In practice, the server may store only the latest state $X[b]$ and the changes induced by the operations with lower sequence numbers. Moreover, once S learns from INVOKE messages that all clients have received and applied all operations with sequence number q, then she may discard the state changes for q as well.

4.3 Client

The clients no longer maintain state s and instead store a digest map $G : \mathbb{N}_0 \to \{0,1\}^*$ indexed by operations, where $G[q]$ authenticates the state resulting from executing the operations with index up to q, starting from s_0. The client uses G to verify the server's responses to his operations in a REPLY message. In particular, for operation o, client C_i runs Algorithm 1, executes its pending-self operations (μ) upon input $G[c]$ to obtain a temporary state a and a corresponding digest g, performs the commutativity check, and, if successful, computes $(d', r') \leftarrow \mathit{verify}_F(g, \alpha, o, r)$. The client halts if the original algorithm halts or if $r' = \perp$; otherwise, the response is $r \leftarrow r'$. The client augments the COMMIT message with α and r' and signs the entire message. Note that d' is again used only temporarily for verifying the pending-self operations and is discarded when the method returns.

Upon receiving a BROADCAST message when the last confirmed operation has index c, the client verifies the signature from client C_j that invoked the operation and the hash value as before. Then C_i intends to verify that the response and digest are consistent (between him and C_j) and to compute the next digest $G[c+1]$. Note that C_i cannot use α, however, to update the digest, as α authenticates o in the state where C_j committed it, but this state may differ from the state at index c, which is current for C_i. We therefore require that S sends an additional authenticator α' for o in state $X[c]$. The client verifies that α' and r correspond to o by executing $(G[c+1], r') \leftarrow \mathit{verify}_F(G[c], \alpha', o, r)$, and verifying that $r' \neq \perp$. The client may garbage-collect entries in G in a similar way as for the hash chain in COP.

5 Conclusion

This paper has introduced COP and ACOP, two variants of the Commutative-Operation verification Protocol, which allow 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); ACOP extends COP by shifting state and operation execution from the clients to the server.

Given the popularity of outsourced computation and cloud computing, the problem of checking the results of remote computations cryptographically has received a lot of attention recently [11, 26, 14, 2]. However, these protocols typically address only a two-party model and, with some exceptions [2], do not support state changes. An important direction for future work lies in integrating these verifiable computation protocols into COP and related protocols for guaranteeing cryptographic integrity in the sense of fork-linearizability for multiple clients.

Acknowledgments. We thank Marcus Brandenburger for interesting discussions and valuable comments.

This work has been supported in part by the European Union's Seventh Framework Programme (FP7/2007–2013) under grant agreement number ICT-257243 TCLOUDS.

References

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

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

[3] Cachin, C.: Integrity and consistency for untrusted services. In: Černá, I., Gyimóthy, T., Hromkovič, J., Jefferey, K., Králović, R., Vukolić, M., Wolf, S. (eds.) SOFSEM 2011. LNCS, vol. 6543, pp. 1–14. Springer, Heidelberg (2011)

[4] Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming, 2nd edn. Springer (2011)

[5] Cachin, C., Keidar, I., Shraer, A.: Fork sequential consistency is blocking. Information Processing Letters 109(7), 360–364 (2009)

[6] Cachin, C., Keidar, I., Shraer, A.: Fail-aware untrusted storage. SIAM Journal on Computing 40(2), 493–533 (2009), preliminary version appears In: Proc. DSN 2009

[7] Cachin, C., Ohrimenko, O.: Verifying the consistency of remote untrusted services with commutative operations. Report arXiv:1302.4808v2, CoRR (December 2013), `http://arxiv.org/abs/1302.4808v2`

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

[9] Canetti, R., Paneth, O., Papadopoulos, D., Triandopoulos, N.: Verifiable set operations over outsourced databases. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 113–130. Springer, Heidelberg (2014)

[10] Clements, A.T., Kaashoek, M.F., Zeldovich, N., Morris, R.T., Kohler, E.: The scalable commutativity rule: Designing scalable software for multicore processors. In: Proc. 24th ACM Symposium on Operating Systems Principles (SOSP), pp. 1–17 (2013)

[11] Cormode, G., Mitzenmacher, M., Thaler, J.: Practical verified computation with streaming interactive proofs. In: Proc. 3rd Conference on Innovations in Theoretical Computer Science (ITCS), pp. 90–112 (2012)

[12] Crosby, S.A., Wallach, D.S.: Authenticated dictionaries: Real-world costs and trade-offs. ACM Transactions on Information and System Security 14(2) (2011)

[13] Feldman, A.J., Zeller, W.P., Freedman, M.J., Felten, E.W.: SPORC: Group collaboration using untrusted cloud resources. In: Proc. 9th Symp. Operating Systems Design and Implementation (OSDI) (2010)
[14] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013)
[15] Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: Double-ended queues as an example. In: Proc. 23rd Intl. Conference on Distributed Computing Systems, (ICDCS) (2003)
[16] Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
[17] Kosba, A.E., Papadopoulos, D., Papamanthou, C., Sayed, M.F., Shi, E., Triandopoulos, N.: TRUESET: Nearly practical verifiable set computations. In: Proc. 23rd USENIX Security Symposium (2014)
[18] Li, J., Krohn, M., Mazières, D., Shasha, D.: Secure untrusted data repository (SUNDR). In: Proc. 6th Symp. Operating Systems Design and Implementation (OSDI), pp. 121–136 (2004)
[19] Li, J., Mazières, D.: Beyond one-third faulty replicas in Byzantine fault-tolerant systems. In: Proc. 4th Symp. Networked Systems Design and Implementation (NSDI) (2007)
[20] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., Walfish, M.: Depot: Cloud storage with minimal trust. In: Proc. 9th Symp. Operating Systems Design and Implementation (OSDI) (2010)
[21] Majuntke, M., Dobre, D., Serafini, M., Suri, N.: Abortable fork-linearizable storage. In: Abdelzaher, T., Raynal, M., Santoro, N. (eds.) OPODIS 2009. LNCS, vol. 5923, pp. 255–269. Springer, Heidelberg (2009)
[22] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica 39, 21–41 (2004)
[23] Mazières, D., Shasha, D.: Building secure file systems out of Byzantine storage. In: Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC) (2002)
[24] Naor, M., Nissim, K.: Certificate revocation and certificate update. IEEE Journal on Selected Areas in Communications 18(4), 561–570 (2000)
[25] Papamanthou, C., Tamassia, R., Triandopoulos, N.: Optimal verification of operations on dynamic sets. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 91–110. Springer, Heidelberg (2011)
[26] Setty, S., Vu, V., Panpalia, N., Braun, B., Blumberg, A.J., Walfish, M.: Taking proof-based verified computation a few steps closer to practicality. In: Proc. 21st USENIX Security Symposium (2012)
[27] Shraer, A., Cachin, C., Cidon, A., Keidar, I., Michalevsky, Y., Shaket, D.: Venus: Verification for untrusted cloud storage. In: Proc. Cloud Computing Security Workshop (CCSW). ACM (2010)
[28] Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003)
[29] Williams, P., Sion, R., Shasha, D.: The blind stone tablet: Outsourcing durability to untrusted parties. In: Proc. Network and Distributed Systems Security Symposium (NDSS) (2009)