Verifying the Consistency of Remote Untrusted Services with Conflict-Free Operations[*]

Christian Cachin[1]
IBM Research - Zurich
cca@zurich.ibm.com

Olga Ohrimenko[2]
Microsoft Research, Cambridge (UK)
oohrim@microsoft.com

8 February 2018

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 *Conflict-free 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. If the server forks two clients by hiding an operation, these clients never again see operations of each other. COP supports *wait-free* client operations in the sense that when executed with a correct server, *non-conflicting* operations can run without waiting for other clients, allowing more parallelism than earlier protocols. A conflict arises when an operation causes a subsequent operation to produce a different output value for the client who runs it. The paper gives a precise model for the guarantees of COP and includes a formal analysis that these are achieved.

Keywords. Cloud computing, fork-linearizability, data integrity, verifiable computation, conflict-free 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 do not trust the provider to always respond correctly, 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, a faulty or *Byzantine* 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 [22] for storage systems. It ensures that whenever the server in her responses to a client C_1 has ignored an operation executed by a client C_2,

[*] A predecessor of this paper with a slightly different title was presented at OPODIS 2014 and appears in the proceedings [8].
[1] Corresponding author. IBM Research - Zurich, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland.
[2] Work done at IBM Research - Zurich and at Brown University.

then C_1 can never again observe an operation by C_2 afterwards and vice versa. In other words, the views of the two clients remain forked after the first inconsistency. This property ensures clearly defined service semantics in the face of an attack and allows clients to detect server misbehavior easily. For instance, the clients may periodically 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.

Several conceptual [9, 21, 6, 7] and practical advances [31, 11, 20, 26, 10] have already improved consistency checking and verification with fork-linearizability and related notions. These protocols ensure that when the server is correct, the service is linearizable and ideally also *wait-free*, that is, every client's operations complete independently of other clients. It has been recognized, however, that *conflicts* between operations can cause a clients to block; this applies to fork-linearizable semantics [22, 9] and to other forking consistency notions [6, 7]. By weakening the notion of fork-linearizability to permit some inconsistent operations in the views of the forked clients, one can circumvent blocking, as illustrated by FAUST [7] and Venus [26].

In this paper, we go beyond storage services and address the consistency of *computation* coordinated by a Byzantine server. The *Conflict-free Operation verification Protocol* or *COP* imposes fork-linearizable semantics for arbitrary functionalities and allows clients to operate concurrently without blocking unless their operations conflict. COP extends earlier protocols aiming at the same goal, in particular, the Blind Stone Tablet (BST) protocol [31]; unlike previous works, COP comes with a detailed formal analysis of its properties.

Supporting wait-free operations and avoiding server-side locks are key features for efficient 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 [22, 9, 18]. For example, two read operations in a storage service never 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 [9]. If *all* operations are to proceed without blocking, though, it is necessary to relax the consistency guarantees to notions such as weak fork-linearizability [7], for instance. COP maintains the stronger property of fork-linearizability and always lets clients proceed at their own speed; conflicting operations that would block are aborted, as considered by Majuntke et al. [21]. The definition of conflicting operations in COP is generic and corresponds to a "write-read conflict" between two concurrent database transactions.

In COP, the server merely coordinates client-side operations but does not compute the responses to operations nor maintain the service state. This conceptually simple approach can be found in many related protocols [31, 12, 11] and practical collaboration systems (such as *git* or *Mercurial* for source-code versioning).

1.1 Contributions

This paper considers a generic service executed by an untrusted server and introduces the Conflict-free Operation verification Protocol (COP) with the following properties:

- COP provides wait-free, abortable consistency verification and ensures fork-linearizability to a group of clients executing an arbitrary joint functionality and using a remote Byzantine server for coordination; it exploits sequences of non-conflicting operations. The notion of conflict considered by COP corresponds to write-read conflicts in databases and generalizes commutative operations used in previous work.
- COP comes with a detailed formal analysis and proof of correctness, showing that it achieves fork-linearizable semantics for generic service emulation; previous work did not establish this notion.

COP follows the general pattern of most previous fork-linearizable emulation protocols, in particular the Blind Stone Tablet (BST) protocol [31]. For determining when to proceed with concurrent operations, we consider whether *sequences* of operations conflict and respect the state of the service, in contrast to earlier protocols, which considered only isolated operations.

COP adopts the notion of conflict-freedom from VICOS [2], which appeared after the preliminary publication of COP. VICOS also illustrates one way to extend COP through authenticated data structures such that the service state is held by the remote server instead of the clients. Combining COP with these appears possible but is left to future work.

1.2 Related work

This section reviews related protocols according to their features and not always in chronological order. A summary of these systems in comparison with COP is given in Table 1.

Storage protocols. Fork-linearizability has been introduced (under the name of *fork consistency*) together with the SUNDR storage system [22, 17]. Conceptually SUNDR operates on storage objects with simple read/write semantics. Subsequent work of Cachin et al. [9] improves the communication cost of untrusted storage protocols to linear in the number of clients, compared to the quadratic overhead of SUNDR. A lock-free storage protocol (called CONCUR) was proposed by Majuntke et al. [21] and introduced the idea of aborting operations that might block. This means that all operations complete in the absence of contention, i.e., when they arrive one after the other at the server; but with contention, all concurrent operations except for one are aborted. CONCUR does not distinguish between types operations that conflict or not.

FAUST [7] and Venus [26] offer only weak fork-linearizability, which excludes the last operation response from the consistency guarantee (i.e., the last operation of one client may differ from the one seen by another client and fork-linearizability holds only after the client completes another operation). However, these two protocols also extend the model by introducing 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 more restrictive model considered here, fork-linearizability is the best possible guarantee [22].

In Depot [20], an even weaker notion called fork-causal consistency is ensured for the stored data by the core algorithm. Through client-to-client communication, Depot also supports "join" operations after forks and achieves a condition called fork-join-causal consistency. This resembles the "eventual consistency" of geo-replicated cloud storage systems, where views may fork temporarily and be reconciled again later. In contrast, COP aims at the stronger notion of fork-linearizability and does not consider communication among clients.

All protocols mentioned so far except Depot use vector clocks (or even vectors of vector clocks) for keeping track of the ordering relation among operations. Depot instead constructs a hash chain over all operations.

Generic services. COP builds on the Blind Stone Tablet (BST) protocol [31], which first extended fork-linearizability consistency verification from storage to generic services. It considers a database hosted by a remote, untrusted server and propagates state updates to all clients after they have been ordered. Every client builds a hash chain over the operation log for keeping track of the consistency with other clients.

BST allows *some* client operations that commute to proceed concurrently and aborts others that do not commute. However, it achieves only limited wait-freedom, even for commuting operations. Furthermore, there is no formal analysis of the consistency achieved by the BST protocol. The discussion of fork-linearizability by Williams et al. [31] only addresses database state updates, but not the responses output by clients. We elaborate on these shortcomings in Section 3.3 and provide in Section 4 a detailed analysis of COP as a key contribution of this work.

Non-blocking protocols. In the context of cloud services replicated over wide-area networks, the tradeoff between fault-tolerance, availability, and consistency has received a lot of attention through the "CAP Theorem" [4]. Services that explicitly allow operations to proceed in parallel whenever possible

Protocol	**Wait-free**	**Function**	**Consistency**	**Execution**	**Proof**
SUNDR [22, 17]	—	storage	fork-lin.	server	—
FAUST [7], Venus [26]	✓	storage	weak fork-lin.	server	✓
CONCUR [21]	—	storage	fork-lin.	clients	✓
BST [31]	(✓)	single commuting op.	(fork-lin.)	clients	—
SPORC [11]	✓	generic op. transform	(weak fork-lin.)	clients	—
Depot [20]	—	storage	fork-join-causal	clients	✓
VICOS [2]	✓	storage, conflict-free	fork-lin.	server	—
COP (this work)	✓	generic, conflict-free	fork-lin.	clients	✓

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 is wait-free for generic *non-conflicting* operation sequences. For the *consistency* property, *weak fork-linearizability* (and *fork-* linearizability*) 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 if this is done by the *server*. Finally, *proof* indicates whether an algorithm has been formally proven correct. Note that VICOS appeared after the initial publication of COP.

provide an attractive way to circumvent the impossibility of being "available" and "consistent" simultaneously in the presence of network "partitions" [25, 24, 27]. Commutative Replicated Data Types (CRDTs) [24], for example, combine strong consistency with immediate responses, in order to enable strong consistency even with replication over wide-area networks. COP exploits a similar, workload-dependent property for achieving a different goal, which allows COP to run on an untrusted server.

SPORC [11] 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 stated to provide fork-* linearizability [18], which is almost the same as weak fork-linearizability [7]; 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 responses of operations.

The subsequent extension of SPORC to Frientegrity [10] leverages read/write operations on storage objects to a complete social network that may be hosted on an untrusted provider. Frientegrity provides fork-* linearizability as consistency condition like SPORC and adds many further features beyond our interest. However, the consistency and integrity verification properties are the same as for SPORC.

FAUST [7] and Venus [26], mentioned before, never block clients and enjoy eventual consistency, but guarantee only weak fork-linearizability.

In contrast to these protocols, COP ensures the stronger fork-linearizability condition, where every operation is consistent as soon as it completes and no client is ever in danger of acting upon receiving an arbitrary output. 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 mandates that all operations are invertible. Therefore SPORC cannot address services with conflicting operations, which exist in many realistic service specifications [9].

VICOS [2] protects the integrity and consistency verification of a generic cloud-object storage service. It extends the protocol of this work and shows how to apply it in a practical deployment.

1.3 Organization and relation to previous version

A predecessor of this paper [8] considered only *commutative* operations instead of conflict-free ones. Brandenburger et al. [2] have subsequently introduced *conflict-free* operations with VICOS and shown that it is sufficient to abort only when a conflict occurs but not for all commuting operations. We take this up here because the notion is more general than commutativity for consistency verification.

Furthermore, the *authenticated* version of COP in [8], which shifted the state from the clients to the server and appeared in the earlier version, is not contained here. The reason lies in the lack of formalization, which would go beyond the scope of this version. VICOS [2] also exploits authenticated data types for keeping the state remotely but does not present a formal consistency analysis. The focus of this work is on a proof for achieving fork-linearizability with COP.

This paper continues by first introducing the notation and basic concepts in Section 2. The subsequent section presents COP and discusses its properties. A detailed analysis of COP follows in Section 4. Finally Section 5 concludes the paper.

2 Definitions

2.1 System model

We consider an asynchronous distributed system with n clients, $C_1, \dots, C_n$ 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.

2.2 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

$$(s', r) \leftarrow F(s, o).$$

This is also called the *sequential specification* of F.

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

$$(s', r) \;=\; F(s_0, \langle o_1, \dots, o_k \rangle)$$

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

2.3 Operations and conflicts

Conflicts between operations of F play an important role in protocols that may execute multiple operations concurrently and have been studied intensively in the context of multi-version concurrency control for databases [30] as well as in concurrent and distributed computing [15]. In this work, an operation $o_1 \in \mathcal{O}$ is said to *conflict* with an operation $o_2 \in \mathcal{O}$ *in a state* $s \in \mathcal{S}$ if and only if the presence of o_1 before o_2 influences the return value of o_2. In other words, if C_1 executes o_1 and o_1 does not conflict with o_2 executed by C_2, then C_2 can go ahead and generate output for o_2 without waiting until o_1 finishes. This improves the throughput of COP compared to earlier protocols.

Conflicts are asymmetric. Formally, o_1 does *not conflict* with o_2 *in a state* s if and only if, for

$$\begin{aligned} (s', r_1) &\leftarrow F(s, o_1) \\ (s'', r_2) &\leftarrow F(s', o_2) \\ (t, q) &\leftarrow F(s, o_2) \end{aligned}$$

it holds

$$r_2 = q.$$

Furthermore, we say that o_1 does *not conflict* with o_2 whenever o_1 does not conflict with o_2 in any state of F. Commuting operations (as considered in earlier work) never conflict, but two non-conflicting operations may not commute.

Not only individual operations, also sequences of them may be conflict-free to each other in this sense. Suppose two sequences μ and ρ consisting of operations in $\mathcal{O}$ are mixed together into one sequence π such that the partial order among the operations from μ and from ρ is retained in π, respectively. If executing π starting from a state s gives the same responses for all operations of ρ as in every other such mixed sequence, in particular $\mu \circ \rho$ and $\rho \circ \mu$, where $\circ$ denotes concatenation, we say that μ does *not conflict* with ρ *in state* s. Analogously, we say that μ does *not conflict* with ρ if μ does not conflict with ρ in any state.

We define a Boolean predicate *conflict*$_F(s, \mu, \rho)$ that returns TRUE if and only if a sequence of operations μ conflicts with a sequence ρ in s according to F. W.l.o.g. we assume that all operations of F and the predicate *conflict*$_F$ are efficiently computable.

Observe that changes to the state of F are not considered in the conflict relation. In particular, two operation sequences μ and ρ that "write" to the same low-level value do not conflict as long as the responses of ρ remain the same as in the absence of μ. This might look different from the usual notion of conflicts considered in other works, because the effects on the underlying state are considered implicitly, only through operations of ρ whose output depends on the state. (Indeed, if μ modifies the state and an operation in ρ returns the complete state, then μ always conflicts with ρ.) As will become clear later, the conflict relation introduced here is adequate for consistency verification.

2.4 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 $F : \mathcal{S} \times \mathcal{O} \to \mathcal{S} \times \mathcal{R}$ to an *abortable* functionality G accordingly. $G : \mathcal{S} \times \mathcal{O} \to \mathcal{S} \times \mathcal{R}_\perp$ 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 response set $\mathcal{R}_\perp = \mathcal{R} \cup \{\perp\}$. G will usually return the same output as F, but it may also return $\perp$ and leave the state unchanged, denoting that a client is not able to execute F. Hence, G is a relation and satisfies

$$G(s, o) = \big\{(s, \perp),\ F(s, o)\big\}.$$

The abortable G inherits most properties of F apart from its deterministic specification. In particular, since G is not deterministic, a sequence of operations no longer uniquely determines the resulting state and response value. Whenever the abortable G is used in a protocol, one has to explicitly require that if G is accessed only sequentially, then G should never abort, i.e., it should always behave like F.

In the sequel, when we refer to a generic *functionality* Λ, this may represent the deterministic F or its abortable extension G.

Abortable functionalities are related to obstruction-free objects [1, 14] in shared-memory systems subject to concurrent operations; such objects also guarantee that every client operation completes assuming the client eventually runs in isolation.

2.5 Operations and histories

The clients interact with a functionality Λ through *operations* provided by Λ. As operations take time, they are represented by two events occurring at the client, an *invocation* and a *response*. A *history* of an execution σ consists of the sequence of invocations and responses of Λ occurring in σ. 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 σ, denoted $o <_\sigma o'$, whenever o completes before o' is invoked in σ. A sequence of events π *preserves the real-time order* of a history σ if for every two operations o and o' in π, 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 σ, the subsequence of σ 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 σ that ends with the last event of o is denoted by $\sigma|^o$.

An operation o is said to be *contained in* a sequence of events σ, denoted $o \in \sigma$, whenever at least one event of o is in σ. 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 [19]). 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 [13].

2.6 Consistency properties

We use the standard notion of *linearizability* [16], which requires that the operations of all clients appear to execute atomically in one sequence. *Fork-linearizability* [22, 9] relaxes the condition of one sequence and extends it 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 operation sequence occurring before that operation is the same. In other words, the history of operations forms a tree whose branches are the forks, and the operations on the path from the root to every leaf are linearizable, and every client observes exactly the operations on the path to one leaf. For simplicity we leave out incomplete operations in (fork-)linearizability; although they could readily be included as in other works, we feel this would overly complicate the analysis of the protocol.

Definition 1 (View). A sequence of events π is called a *view* of a history σ at a client C_i w.r.t. a functionality Λ if:

1. π is a sequential permutation of some subsequence of complete operations in σ;
2. all complete operations executed by C_i appear in π; and
3. π satisfies the sequential specification of Λ.

Definition 2 (Linearizability [16]). A history σ is *linearizable w.r.t. a functionality* Λ if there exists a sequence of events π such that:

1. π is a view of σ at all clients w.r.t. Λ; and
2. π preserves the real-time order of σ.

Definition 3 (Fork-linearizability [22]). A history σ is *fork-linearizable w.r.t. a functionality* Λ if for each client C_i there exists a sequence of events π_i such that:

1. π_i is a view of σ at C_i w.r.t. Λ;
2. π_i preserves real-time order of σ; 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* [9]. It summarizes the requirements put on our protocols, 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 [9]). We say that a protocol P for a set of clients *emulates* a functionality Λ *on a Byzantine server S with fork-linearizability* if:

1. in every fair and well-formed execution of P, the sequence of events observed by the clients is fork-linearizable with respect to Λ; and
2. if S is correct, then the execution is linearizable w.r.t. Λ.

2.7 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, $sign_i$ and $verify_i$. The invocation of $sign_i$ specifies the index i of a client and takes a bit string $m \in \{0,1\}^*$ as input and returns a signature $\sigma \in \{0,1\}^*$ with the response. Only C_i may invoke $sign_i$. The operation $verify_i$ takes a putative signature σ and a bit string m as parameters and returns a Boolean value with the response. Its implementation satisfies that $verify_i(\sigma, m)$ returns TRUE for any $i \in \{1, \ldots, n\}$ and $m \in \{0,1\}^*$ if and only if C_i has executed $sign_i(m)$ and obtained σ before; otherwise, $verify_i(\sigma, m)$ returns FALSE. Every client as well as S may invoke *verify*. The signature scheme may be implemented analogously to the hash function.

3 The conflict-free operation verification protocol

3.1 Protocol description

Notation. The function $length(L)$ for a list L denotes the number of elements in L and $||$ stands for the concatenation of strings. Several variables are *dynamic arrays* or *maps*, which associate keys to values. A value v is stored in a map H by assigning it to a key k, denoted $H[k] \leftarrow v$; if no value has been assigned to a key, the map returns $\perp$. For simplicity, $\perp$ also stands for the empty bit string. Recall that G is the abortable extension of functionality F.

Overview. The pseudocode of COP for the clients and the server is presented in Algorithms 1–2. We assume that the execution of each client is well-formed and fair.

COP adopts the structure of previous protocols that guarantee fork-linearizable semantics [22, 31, 5]. It aims at obtaining a globally consistent order for the operations of all clients, as determined by the server. Every client maintains a copy of the service state and executes all operations locally.

When a client C_i *invokes* an operation o, he sends an INVOKE message to the server S (L7–L10). 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 which go beyond the prefix of the history that C_i has already verified for consistency. These pending operations are ordered before o by a correct S, but C_i may not yet know about some of them. (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.

When C_i receives the REPLY message with o, he verifies whether the data from the server is consistent and, if everything is valid, he commits o. COP uses **assert** statements for verification. If any of these steps fail, 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 executes o by *simulating* the pending-self operations and o according to F (that is, without updating the locally held state). If the pending-other operations do *not conflict* with the pending-self operations and o, then he declares o to be *successful* and outputs the response r according to F, as resulting from the simulated operations. Otherwise, the client *aborts* o and the response is $r = \bot$. 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 (L106–L110). 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 (L40–L47).

Data structures. Every client locally maintains a set of variables during the protocol (L1–L6). The state $s \in \mathcal{S}$ is the result of applying all confirmed and 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 operation sequence as announced by S to C_i. 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 hash chain allows for fast comparisons between the histories of two clients: if they obtain the same hash-chain value, then both have confirmed the same sequence of operations.

The client sets a variable $\bar{o}$ to o whenever it has invoked an operation o but not yet completed it; at other times $\bar{o}$ is $\bot$. Variable Z maps the sequence number of every operation that the client has executed himself to its status. The client only needs the entries in Z with indices greater than c.

The (correct) server also keeps several variables locally (L101–L105). Variable t determines the global sequence number for the invoked operations and b denotes the sequence number of the last broadcast operation. The latter ensures that S disseminates operations to clients in the global order. Furthermore, she stores the invoked operations in a map I and the completed operations in a map O, both indexed by sequence number.

Protocol. When client C_i invokes an operation o (L7–L10), he stores it in $\bar{o}$ and sends an INVOKE message to S containing o and τ, a digital signature computed over o and i. In turn, a correct S sends a REPLY message with the list *Pend* of pending operations (L106–L110); the operations have sequence numbers $c+1, c+2, \ldots$. Upon receiving a REPLY message, the client checks that *Pend* is consistent with any previously sent operations and uses *Pend* to assemble the pending-other operations *Pend-other* and the successful pending-self operations *Pend-self*. He then determines whether o can be executed or has to be aborted (L11–L39).

In particular, during the loop in Algorithm 1 (L15–L27), for every operation o in *Pend*, client C_i determines its sequence number l and verifies from the INVOKE signature that o was indeed invoked by C_j (L17–L18). He computes the entry of o in the hash chain from o, l, j, and $H[l-1]$. If $H[l] = \bot$, 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 (L19–L22).

If operation o is his own and its saved status in $Z[l]$ was SUCCESS, then he appends it to *Pend-self* (L23–L24). The client remembers the status of his own operations in Z, since $\mathit{conflict}_F$ depends on the

Algorithm 1 Conflict-free operation verification protocol (client C_i)

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

7: upon invocation $o$ do                                                // invoke operation $o$
8:     $\bar{o} \leftarrow o$
9:     $\tau \leftarrow sign_i(\text{INVOKE}\|o\|i)$
10:    send message $[\text{INVOKE}, o, \tau]$ to $S$

11: upon receiving message $[\text{REPLY}, Pend]$ from $S$ do            // the last operation in Pend should be $\bar{o}$
12:    $Pend\text{-}other \leftarrow \langle\rangle$                      // list of pending-other operations
13:    $Pend\text{-}self \leftarrow \langle\rangle$                       // list of successful pending-self operations
14:    $k \leftarrow 1$
15:    while $k \leq length(Pend)$ do
16:        $(o, j, \tau) \leftarrow Pend[k]$
17:        $l \leftarrow c + k$                                          // promised sequence number of $o$
18:        assert $verify_j(\tau, \text{INVOKE}\|o\|j)$
19:        if $H[l] = \perp$ then
20:            $H[l] \leftarrow hash(H[l-1]\|o\|l\|j)$                    // extend hash chain
21:        else
22:            assert $H[l] = hash(H[l-1]\|o\|l\|j)$                     // server replies must be consistent
23:        if $j = i \wedge k < length(Pend) \wedge Z[l] = \text{SUCCESS}$ then
24:            $Pend\text{-}self \leftarrow Pend\text{-}self \circ \langle o\rangle$
25:        else if $j \neq i$ then
26:            $Pend\text{-}other \leftarrow Pend\text{-}other \circ \langle o\rangle$
27:        $k \leftarrow k + 1$
28:    // variables $o$, $j$, and $l = c + length(Pend)$ keep their values
29:    assert $k > 1 \wedge o = \bar{o} \wedge j = i$                    // last pending operation must equal the current operation
30:    if not $conflict_F(s, Pend\text{-}other, Pend\text{-}self \circ \langle o\rangle)$ then   // $o = \bar{o}$ is the current operation
31:        $(s', r) \leftarrow F(s, Pend\text{-}self \circ \langle o\rangle)$   // compute response to $o$ and ignore resulting state
32:        $Z[l] \leftarrow \text{SUCCESS}$
33:    else
34:        $r \leftarrow \perp$
35:        $Z[l] \leftarrow \text{ABORT}$
36:    $\phi \leftarrow sign_i(\text{COMMIT}\|o\|l\|H[l]\|Z[l])$             // commit operation $\bar{o}$
37:    send message $[\text{COMMIT}, o, l, H[l], Z[l], \phi]$ to $S$
38:    $\bar{o} \leftarrow \perp$
39:    return $r$                                                       // complete operation $\bar{o}$

40: upon receiving message $[\text{BROADCAST}, o, l, h, z, \phi, j]$ from $S$ do
41:    assert $l = c + 1 \wedge verify_j(\phi, \text{COMMIT}\|o\|l\|h\|z)$   // start to confirm operation $o$
42:    if $H[l] = \perp$ then                                           // operation $o$ has never been pending at $C_i$
43:        $H[l] \leftarrow hash(H[l-1]\|o\|l\|j)$
44:    assert $h = H[l]$                                                // if this holds, then $o$ is confirmed
45:    if $z = \text{SUCCESS}$ then                                     // apply $o$ only if successful
46:        $(s, r') \leftarrow F(s, o)$                                  // apply operation $o$ and ignore response
47:    $c \leftarrow c + 1$
```

Algorithm 2 Conflict-free operation verification protocol (server S)

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

106: upon receiving message $[\textsc{invoke}, o, \tau]$ from $C_i$ do
107:     $t \leftarrow t + 1$
108:     $I[t] \leftarrow (o, i, \tau)$
109:     $Pend \leftarrow \langle I[b+1], \ldots, I[t] \rangle$          // include non-committed operations and $o$
110:     send message $[\textsc{reply}, Pend]$ to $C_i$

111: upon receiving message $[\textsc{commit}, o, l, h, z, \phi]$ from $C_i$ do
112:     $O[l] \leftarrow (o, h, z, \phi, i)$
113:     while $O[b+1] \neq \perp$ do          // broadcast operations ordered by their sequence number
114:         $b \leftarrow b + 1$
115:         $(o', h', z', \phi', j) \leftarrow O[b]$
116:         send message $[\textsc{broadcast}, o', b, h', z', \phi', j]$ to all clients
```

state and that could have changed if he applied operations after committing o. Operations of other clients from *Pend* are added to *Pend-other* (L26).

Finally, when C_i reaches the end of *Pend*, he checks that *Pend* is not empty and that it contains $o = \bar{o}$ at the last position (L29). He then tests whether the pending-other operations *Pend-other* do not conflict with *Pend-self* $\circ \langle o \rangle$ in state s, his state resulting from the confirmed operations (L30). If there is no conflict, he records the status of o as SUCCESS in $Z[l]$ and computes the response r by executing *Pend-self* $\circ \langle o \rangle$ starting from s (L31–L32). Otherwise, if *Pend-other* conflicts with o, he records the status of o as $Z[l] \leftarrow$ ABORT and sets $r \leftarrow \perp$ (L33–L35). Then C_i signs o together with its sequence number, status, and hash chain entry $H[l]$, includes all values in the COMMIT message sent to S, and returns r (L36–L39). Through these steps C_i *commits o*.

Upon receiving a COMMIT message for an operation o with sequence number l, the (correct) server records its content as $O[l]$ in the map of committed operations (L112). Then she is supposed to send a BROADCAST message containing $O[l]$ to the clients. She waits with this until she has received COMMIT messages for all operations with sequence number less than l and has also broadcast them (L113–L116). This ensures that completed operations are disseminated in the global order to all clients, exploiting the FIFO channels between the correct S and the clients.

In a BROADCAST message received by client C_i (L40), the committed operation is represented by a tuple (o, l, h, z, ϕ, j). Client C_i conducts several verification steps. If successful, we say o is *confirmed*. If o did not abort, then C_i subsequently *applies* o to his state s. In more detail, the client first verifies that the sequence number l is the next operation according to c (L41); hence, o follows the global order and the server did not omit any operations. Second, he uses the COMMIT signature ϕ in the message to verify that C_j indeed committed o (L41). Lastly, C_i computes his own hash-chain entry $H[l]$ for o and asserts that it is equal to the hash-chain value h from the message (L42–L44). 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 (L45–L46).

Observe that C_i must output the response of a successful operation after receiving the REPLY message (L39). To satisfy fork-linearizability for the output, the view of C_i must contain at least its pending-self operations and the output value must not change even if a faulty server would cause the other clients to commit the pending-other operations differently than announced to C_i. The state (s' in L31) computed by C_i is ignored though, as it is computed with the pending-other operations skipped (hence, it may not reflect all operations in C_i's view).

3.2 Features of COP

Conflicts in operation sequences. Consider the following example F of a counter restricted to non-negative values: Its state consists of an integer s; an $\mathit{add}(x)$ operation adds x to s and returns TRUE; a $\mathit{dec}(x)$ operation subtracts x from s and returns TRUE if $x \leq s$, but does nothing and returns FALSE if $x > s$.

We use this to illustrate three properties of COP, where S is always correct in the examples. Assume all client operations have completed and that the state (after applying all operations) at C_i is $s = 7$.

1. Suppose C_i executes $\mathit{add}(3)$ and the REPLY message contains a pending operation $\mathit{dec}(10)$. The operation of C_i succeeds and is executed because no *add* or *dec* operation conflicts with $\mathit{add}(3)$, as its response is always TRUE. However, the operations $\mathit{add}(3)$ and $\mathit{dec}(10)$ do not commute in state 7 because the response from $\mathit{dec}(10)$ differs in the two possible orderings (the resulting states also differ, but this is not relevant for our notion of a conflict). This shows that testing for *conflict-free operations* permits more executions to succeed than checking only commuting operations. Hence, COP aborts fewer executions than protocols aborting all non-commuting operations.
2. Client C_i executes $\mathit{dec}(5)$ and subsequently $\mathit{dec}(4)$, while $\mathit{add}(3)$ by another client is pending at both times. Note that C_i executes $\mathit{dec}(5)$ successfully but aborts $\mathit{dec}(4)$ because $\mathit{add}(3)$ conflicts with $\langle \mathit{dec}(5), \mathit{dec}(4) \rangle$ in state 7.
   However, considering C_i's operations individually, $\mathit{add}(3)$ does not conflict with $\mathit{dec}(5)$ nor with $\mathit{dec}(4)$ in state 7 because their return values are the same when $\mathit{add}(3)$ is omitted (although the resulting states differ). This shows why the client considers the *sequence* of all successful *pending-self* operations when testing for a conflict with the current operation.
3. Suppose that C_i executes $\mathit{dec}(5)$ and S reports the pending sequence $\langle \mathit{dec}(2), \mathit{dec}(1) \rangle$. Thus, C_i aborts $\mathit{dec}(5)$. Although, when considered individually from state 7, $\mathit{dec}(2)$ does not conflict with $\mathit{dec}(5)$ and $\mathit{dec}(1)$ neither conflicts with $\mathit{dec}(5)$, their concatenation conflicts with $\mathit{dec}(5)$ and thus C_i aborts. This illustrates why COP checks for a conflict between the *sequence* of *pending-other* operations and the target operation.

Neither of these three properties is present in previous protocols (as also discussed in Section 3.3).

Memory requirements. For saving 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 in memory only the last applied operation and the pending operations in H and the 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.

Complexity. In terms of *communication* cost, 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 sends a BROADCAST message to all clients. Thus, when ℓ operations are executed overall, the protocol basically takes $O(\ell n)$ messages, although subsequent broadcasts to the same client could be batched until the client invokes the next message [2]. 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 *Pend* of pending operations. If even one client is slow, then the length of *Pend* 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.

The total computational cost, on the other hand, is $O(\ell n)$ for executing ℓ operations of F, and this cannot be reduced as easily. The reason lies in the maintenance of the hash chain at all clients, which must be updated for every operation. Moreover, if a large number of pending operations are present during an operation, the verification cost of the client increases proportionally.

Aborts and wait-freedom. Every client executing COP may proceed with an operation o for F as long as no pending operations of other clients conflict with o. Observe that the response to o obtained by the client 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 directly while processing the REPLY message from S. However, when the pending operations of other clients conflict with o, the response would differ. Thus, the client aborts o and outputs $\perp$ according to G.

Hence, for F where no operations or operation sequences conflict COP is wait-free; in particular, this holds when all of them commute. For arbitrary F, however, no fork-linearizable Byzantine emulation can be wait-free [9]. COP avoids blocking via the augmented functionality G. Clients complete every operation in the sense of G, which includes aborts; therefore, COP is wait-free for G. 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 [17, 26]; these methods apply also to COP.

3.3 Comparison to Blind Stone Tablet (BST)

The BST protocol [31] is a direct predecessor of COP but has several shortcomings and does not achieve all claimed properties, as explained now.

BST considers transactions on a database, coordinated by the remote server. A client first simulates a transaction using his own copy, potentially generating local output, then undoes this transaction on his copy, and coordinates with the server for committing the transaction. From the server's response he determines if a transaction individually commutes with every other, pending transaction that was reported by the server as invoked by different clients. If there is a conflict, the client "rolls back the external effects" of the transaction and basically aborts; otherwise, he "commits" the transaction (but without changing his database copy) and relays it via the server to other clients. When a client receives such a relayed transaction, he applies the transaction to his database copy. At this high level BST is similar to COP.

However, when considering the details, several limitations of BST become apparent: First, a client applies his own transactions only after all pending transactions by other clients have been applied to his own database copy. This means that when the client executes a transaction T_B, updates induced by an earlier transaction T_A of his may not yet be reflected in the database copy because they may be held back by earlier transactions of other clients, which were pending during the execution of T_A, but have not yet been applied. Thus, the client might execute T_B (in the simulation step) from a *wrong* state, and this may yield incorrect output for a linearizable execution. Checking for the absence of conflicts between T_B and other transactions may also use such a faulty state. Alternatively, the protocol should block until the changes from T_A are applied to the database copy, but then the protocol is no longer "wait-free" as stated [31].

Second, the BST client checks conflicts between his current transaction and the incoming (pending) ones individually, considering each one alone but not as the intended execution sequence. Like the first limitation, this implies that the client could violate consistency. In particular, the second and third example executions above, used for illustrating the conflicts in Section 3.2, will fail and produce wrong outputs in BST.

Third, the notion of "trace consistency" in the analysis of BST considers only the *database state* and the transactions that have been *executed* on the local state [31]. However to satisfy fork-linearizability one must consider the responses output by the client. The formal notion of linearizability does not even consider the state of a functionality, only the views of the clients are relevant. A transaction may be applied long after the client received the response and acted on it. Hence, proof sketch available for BST [31, Sec. 5.2] does not establish fork-linearizability.

COP extends BST and allows one client to execute multiple operations without waiting, i.e., independently of the speed of other clients, as long as the *sequence* of pending operations by other clients jointly does not conflict with the client's operations, considering the current service state. Moreover, the analysis of COP shows it is fork-linearizable for all *responses* output by clients.

4 Analysis

This section establishes the key properties of conflict-free operation verification protocol (COP) in Algorithms 1–2.

The first theorem addresses executions with a correct server and its proof appears in the next section. For stating the this result, we define the following notion of overlapping operations. It is a refinement of a sequential execution that additionally takes into account the event that a client *applies* one of *its own* operations (L40–L47); we introduce the term that the operation is *self-applied* to denote the event that this occurs. We say that two operations o and o' in a history σ *overlap* whenever the invocation of o occurs after o' is invoked and before o' is self-applied in σ, or vice versa, the invocation of o' occurs after the invocation of o is invoked and before o is self-applied. An execution σ *without overlaps* is one in which no two operations overlap.

Theorem 1. *If the server is correct, then the history of every execution of COP is linearizable w.r.t. the abortable functionality G. Furthermore, if the clients execute all operations without overlaps, then all histories of COP are linearizable w.r.t. F and no operations abort.*

The second theorem addresses executions with a Byzantine server and captures the key goal of COP.

Theorem 2. *In every well-formed execution of COP, the history of events observed by the clients is fork-linearizability w.r.t. the abortable functionality G.*

Together these results imply our main result. Recall that a Byzantine emulation implies that the execution is linearizable when S is correct and that it is fork-linearizability otherwise.

Corollary 3. *COP emulates the abortable functionality G on a Byzantine server with fork-linearizability.*

In the analysis we use the following terminology. When a client issues a COMMIT signature for some operation o, we say that he *commits o*. The client's sequence number included in the signature thus becomes the *sequence number of o*; note that with a faulty S, two different operations may be committed with the same sequence number by separate clients.

4.1 Operating with a correct server

This section contains a proof for Theorem 1, which assumes S is correct. In particular, we show that the output of every client satisfies G also in executions with concurrent or overlapping operations. The check for conflicts, applied after simulating the client's pending-self operations, ensures that the client's response remains unchanged regardless of whether the pending-other operations execute before the operation itself or not.

Lemma 4. *If the server is correct, then every history σ is linearizable w.r.t. G.*

Proof. Recall that σ consists of invocation and response events. We now explain how to construct a sequential permutation π of σ. We often rely on the correspondence between the pair of invocation and response events of one operation in σ and the operation itself; hence, we sometimes treat π as a sequence of operations to simplify the terminology.

For the construction of π, note that a client sends an INVOKE message with his operation o to the server (L10), the server assigns a sequence number to o, and sends it back (L106–L110). Since S is correct, this is also the sequence number of o. The client then computes the response and sends a signed COMMIT message to S, containing the operation and its sequence number, and also outputs the response (L11–L39). Let π consist of all events in σ, ordered first by the sequence number of the corresponding operation and including the invocation before the response with the same sequence number.

As the server is correct, she processes 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, π preserves the real-time order of σ, which is the second property of linearizability.

We now show the first property of linearizability, i.e., that π is also a view of σ for all clients w.r.t. G. The sequence π is a view of σ at a client C_i if it satisfies three conditions (Definition 1). The first conditions holds because π is constructed as a permutation of σ. Since each executed operation appears in σ in terms of its invocation and response events, π contains all operations of all clients. This implies the second condition of a view. It remains to show that π satisfies the sequential specification of G.

For reasoning about G, we introduce additional notation to capture the fact that it is not deterministic. For a sequence ω of operations of G occurring in an actual execution, we write *successful*(ω) for the subsequence whose status was SUCCESS, determined for each operation by the client that executed the operation. Restricted to successful operations, G is deterministic and reduces to F.

In particular, consider some operation $o \in \pi$, executed by client C_i and fix a schedule that determines which operations are successful. We want to show the following claim:

> *For any client C_j (including $C_j = C_i$), the tuple $(s, r') \leftarrow F(s, o)$ computed when C_j applies o in L46 satisfies:*
>
> 1. $(s, r') = F(s_0, \textit{successful}(\pi|^{o}))$;
> 2. *If $o \in$ successful$(\pi|^{o})$, i.e., if o is successful, then r' is equal to the response r that C_i has output when it completed o (L39); otherwise, C_i has responded with $\perp$*

We use induction on the operations sequence π to show this.

Consider the base case where o is the first operation in π and recall that every client initializes its local state variable s to s_0. 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 SUCCESS, computes $(s', r) \leftarrow F(s_0, o)$ and outputs r. When C_j later receives o in the BROADCAST message from S with sequence number 1, he applies o because he learns status of o in z. Then C_j updates the state s as $(s, r') \leftarrow F(s_0, o)$. Since F is deterministic, $(s, r') = (s', r)$ and the claim follows.

Now consider the case when o is not the first operation in π and assume that the induction assumption holds for the operation that appears in π before o. If the status of o is ABORT, then o is filtered out by the *successful*() operator in the claim; similarly, C_j leaves the state s unchanged upon applying o (L45–L46). In addition, C_i has responded with $\perp$ since o was aborted (L30–L35). The claim follows.

Otherwise, if o succeeds, we need to show that the state s at client C_j after applying o satisfies (s, r) $= F(s_0, \textit{successful}(\pi|^{o}))$ and that the response of C_i is $r \neq \perp$. Since S is correct, she assigns unique sequence numbers to the operations in the order in which she receives them in INVOKE messages (L106–L110). According to the code for confirming and applying operations, C_i therefore processes (via L41) a sequence of operations that is a prefix of π, takes into account the status of each operation, and filters out those that abort (L45–L46). This ensures the first property of the claim.

Let ρ be the sequence of operations that C_i has confirmed before he received the REPLY containing o; this sequence is in the order of the sequence numbers assigned by S and in the order in which C_i confirmed these operations. It follows from the construction of π that $\rho = \pi|^{o^*}$, where o^* is the last operation in ρ. The induction assumption implies that variable s at C_i after applying o^* is equal to s^*, defined by

$$(s^*, \cdot) = F(s_0, \textit{successful}(\pi|^{o^*})) = F(s_0, \textit{successful}(\rho)). \tag{1}$$

Thus, C_i starts processing the REPLY message for o containing the list *Pend* from state $s = s^*$ (L11–L39). C_i constructs implicitly a permutation *Pend-self* $\circ \langle o \rangle \circ$ *Pend-other* of *Pend*. Recall that we consider the case where operation o succeeds and *Pend-other* does *not* conflict with *Pend-self* $\circ \langle o \rangle$ in state s^*, as ensured in L30. Thus, C_i outputs response r given by $(\cdot, r) \leftarrow F(s^*, \textit{Pend-self} \circ \langle o \rangle)$ in L39. The definition of non-conflicting operation sequences implies that r is also equal to the response $\bar{r}$ from

$$(\cdot, \bar{r}) = F(s^*, \textit{Pend-other} \circ \textit{Pend-self} \circ \langle o \rangle)$$

because this is a mixed operation sequence from *Pend-other* and *Pend-self* $\circ \langle o \rangle$, which preserves the partial order of operations from the subsequences.

It follows first from the construction of *Pend-other* and *Pend-self*, which contain all operations of *Pend* except for o and the aborted pending-self operations of C_i, second, from their conflict-freedom, and, third, from recalling that *Pend-self* does not contain aborted pending-self operations that also the response $\tilde{r}$ from

$$(\cdot, \tilde{r}) = F(s^*, \textit{successful}(\textit{Pend})) \tag{2}$$

satisfies $r = \bar{r} = \tilde{r}$.

The definition of F on operation sequences implies, furthermore, that

$$F(s^*, \textit{successful}(\textit{Pend})) = F(s_0, \textit{successful}(\pi|^o) \tag{3}$$

because o is the last operation in *Pend* and according to the definition of s^* in (1).

To show that $r = r'$, where r' is computed by C_j when he applies o (L46), note that C_j has applied all successful operations in π up to o at this time and computed $(s, r') = F(s_0, \textit{successful}(\pi|^o)$. Combining this with (3) and (2) now shows that $r' = r$ and the third property of a view follows.

Note that the claim holds for any client C_j, therefore, σ is linearizable w.r.t. G. □

Lemma 5. *If the clients execute all operations without overlaps, then all histories of COP are linearizable w.r.t. F and no operations abort.*

Proof. Consider an operation o that a client C_i has invoked and suppose towards a contradiction that it aborts. According to the protocol, this occurs only if *Pend* in the REPLY message with o to C_i contains some pending-other operation, say, o' executed by C_j. This implies that o' has not been applied yet by C_i, even though C_j has applied it according to the assumption that the execution does not have any overlapping operations.

However, because C_j has applied o' and S is correct, it follows that S has also sent the BROADCAST message containing o' to C_j earlier, before C_j has applied o'. Note that messages between the correct server and one client are delivered in FIFO order. Hence, C_i receives the BROADCAST message corresponding to o' and has applied o' *before* processing the REPLY message containing o. This implies that $o' \notin$ *Pending* according to server's operation (L106–L110). Hence, C_i does not abort, which contradicts the assumption. □

4.2 The promised view of an operation

In this and the next section, we prove Theorem 2. The proof starts by constructing 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. Then, in order to be fork-linearizable even if C_i will not confirm o later, 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 (Lemma 7). The view of C_i further includes all operations that C_i has executed after o. Taken together this will demonstrate that every execution of COP is fork-linearizable w.r.t. G (Lemma 12).

Suppose a client C_i executes and thereby commits an operation o. We define the *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 *Pend* of pending operations received in the REPLY message during the execution of o, including o itself (according to the protocol C_i verifies that the last operation in *Pend* is o).

The protocol constructs a hash chain H over a sequence of (index, operation, client)-triples of the form $(1, o_1, i_1), \ldots, (l, o_l, i_l)$. Starting from $H[0] = \perp$, we set $H[k] \leftarrow \textit{hash}(H[k-1] \| o_k \| k \| i_k)$ for $k = 1, \ldots, l$. The value $h = H[l]$ at the tip of the hash chain *represents* the operation sequence $\langle o_1, \ldots, o_l \rangle$. According to the collision-resistance of the hash function, no two different operation sequences are represented by the same hash value.

Lemma 6. *After C_j has confirmed some operation o at index l, his hash-chain value $H[l]$ represents the sequence of operations that he has confirmed up to o.*

Proof. Recall that C_j extends H in two places: when he confirms an operation at some index l (L42–L43), and when he receives a REPLY message with pending operations (L19–L20). According to the checks when C_j receives an operation to confirm in a BROADCAST message (L41), the client builds the hash chain H incrementally, controlled by variable c, in the sequence of the operations that he confirms. An operation o' from *Pend*, at some index l' higher than c, might also have been inserted into H within the loop (L15–L27) earlier, when C_j executes an operation of his own. This is also controlled by c (L17). But when C_j later receives a BROADCAST message with this index l', any operation o^*, and any hash-chain tip h, he verifies the COMMIT-signature (L41) and checks that the hash-chain entry $H[l'] = \textit{hash}(H[l'-1] \| o^* \| l' \| i)$ computed by himself is equal to the signed h (L44). Since this succeeds, $o' = o^*$ and $H[l]$ represents the sequence of operations that C_j has confirmed up to o. $\square$

Lemma 7. *If C_j has confirmed some operation o that was committed by a client C_i (including $C_i = C_j$), 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 clients C_j and C_k have confirmed an operation o committed by C_i, then C_j and C_k have both confirmed the same sequence of operations up to o;*
2. *the promised view to C_i of o contains all operations executed by C_i up to o.*

Proof. We first investigate the promised view to C_i of o, which by definition consists of the sequence of operations that C_i has confirmed, followed by the list *Pend* in the REPLY message, including o. Consider the time when C_i receives the REPLY message during the execution of o. We first show that when C_i commits o with sequence number l, the hash-chain entry $H[l]$ represents the promised view to C_i of o.

According to Lemma 6, $H[c]$ represents the sequence of operations confirmed by C_i so far. For every pending operation $p \in$ *Pend*, 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 *Pend*. 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 and stored in $H[l]$ (L22). Later, C_i verifies that o itself has also been returned to him as pending (L29). Hence, the new hash value h stored in H at the sequence number of o (i.e., $H[l]$ at L36) represents the promised view to C_i of o. Then C_i issues a COMMIT signature ϕ on o and h and sends ϕ to the server.

When C_j receives the BROADCAST message from S with the COMMIT-signature ϕ of C_i and operation o to be confirmed and applied by C_j with sequence number l, he verifies the COMMIT-signature of C_i on o, l, and h, and only confirms o if the hash value satisfies $h = H[l]$ (L44). Recall from Lemma 6 that $H[l]$ represents the sequence of operations that C_j has confirmed up to o. Noting that the hash function has no collisions, h and $H[l]$ represent the same sequence of operations and the main statement of the lemma follows.

The first additional claim follows by applying the lemma twice for o committed by C_i. For showing the second additional claim, we note that if C_i confirms an operation by himself, then he has previously executed it. 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 *Pend* from the REPLY message. Thus, they are also in the promised view of o. □

4.3 The view of a client

We construct a sequence π_i from σ 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). Let α_i be a sequence of operations constructed as follows. It contains all operations confirmed by C_k up to and including o; if C_i has confirmed o, then append the operations that C_i has confirmed after o (if any).

Furthermore, let β_i be the sequence of operations committed by C_i with a sequence number higher than that of o. Then π_i is the concatenation of α_i and β_i.

Observe that by definition, every operation in α_i has been confirmed by *some* client and *no* client has confirmed operations from β_i.

Lemma 8. *The sequence π_i is a view of σ at C_i w.r.t. G.*

Proof. Note that π_i is defined through a sequence of operations that are contained in σ. Hence π_i is sequential by construction.

We now argue that all operations executed by C_i are included in π_i. Recall that $\pi_i = \alpha_i \circ \beta_i$ and consider o, the last operation of C_i in α_i. As o has been confirmed by some C_k, Lemma 7 shows that α_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 π_i all other operations executed by C_i are contained in β_i, and the property follows.

The last property of a view requires that π_i satisfies the sequential specification of G. Note that G is not deterministic and some responses might be $\perp$. But when we ensure that two operation sequences of G have responses equal to $\perp$ 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 α_i and assume no operation aborts and returns $\perp$. Consider any $o_j \in \alpha_i$, executed by a client C_j (including $C_i = C_j$). Lemma 7 implies that $\alpha_i|^{o_j}$ is a prefix of the promised view to C_i of o. We want to show that the response r_j of o_j to C_j satisfies the specification of G, i.e., that $(\cdot, r_j) = F(s_0, \textit{successful}(\alpha_i|^{o_j}))$.

For the point in time when C_j executes o_j, define ρ_j to be the sequence of operations that C_j has confirmed prior to this and define s_j to be the state resulting from applying the successful operations in ρ_j, as stored in variable s. This implies that $(s_j, \cdot) = F(s_0, \textit{successful}(\rho_j))$ according to the protocol (L40–L47), and using the notation $\textit{successful}(\cdot)$ from Lemma 4.

We want to show that the response of o_j to C_j satisfies G as well. Let *Pend* be the pending operations contained in the REPLY message from S to C_j. Observe that C_j partitions *Pend* into *Pend-other* (pending-other operations), *Pend-self* (successful pending-self operations), and the aborted pending-self operations of C_j, where o_j is also among the pending-self operations.

Client C_j then checks if *Pend-other* does not conflict with *Pend-self* $\circ \langle o_j \rangle$ in s_j (L30), and if this is the case (L31), C_j computes the response r_j for o_j from state s_j as $(\cdot, r_j) \leftarrow F(s_j, \textit{Pend-self} \circ \langle o_j \rangle)$. Since *Pend-self* and *Pend-other* preserve the relative order of operations in *Pend*, the definition of non-conflicting operations implies that the responses of C_j from $F(s_j, \textit{successful}(\textit{Pend}))$ and $F(s_j, \textit{Pend-self} \circ \langle o \rangle)$ are equal. This demonstrates that $(\cdot, r_j) = F(s_j, \textit{successful}(\textit{Pend})) = F(s_0, \textit{successful}(\alpha_i|^{o_j}))$, i.e., that o_j satisfies the sequential specification of G assuming no aborts. Recall this holds for any operation o_j in α_i and that $\pi_i = \alpha_i \circ \beta_i$.

To conclude the argument, we still have to show that the abort status for every operation $o_j \in \alpha_i$ is the same for any client C_k (including $C_k = C_i$) who confirms o_j, and C_j (who has committed o_j). Then they will produce the same responses and same state. Note that when C_j executes o_j, he either computes a response according to F or aborts the operation, declaring its status to be SUCCESS or ABORT, respectively (L30–L35). The status z is signed, sent to S in the COMMIT message, and should be received in the BROADCAST message (L40) by C_k. Since C_k has confirmed o_j, he has verified the COMMIT signature and this implies that the status taken into account by C_k is also equal to z, as used to determine whether he updates the state with o_j (L41–L46).

Furthermore, we need to show that the operations in β_i satisfy the specification of G, where β_i consists of operations committed by C_i with a sequence number higher than that of o. According to the earlier argument about α_i and considering that C_i has confirmed o when computing the responses of operations in β_i, when C_i receives the REPLY message for o (L11), its state s results from confirming and applying all operations in α_i. Hence, $(s, \cdot) = F(s_0, \mathit{successful}(\alpha_i))$. For every successful $o_i \in \beta_i$ client C_i computes the response r_i of o_i (L31) as $(\cdot, r_i) = F(s, \mathit{Pend\text{-}self} \circ \langle o_i \rangle)$, where it is easy to see that the variable *Pend-self* (which does not contain aborted operations) is a prefix of $\mathit{successful}(\beta_i)$. Since C_i executes and commits the operations of β_i in the order of their sequence numbers, it follows that also $(\cdot, r_i) = F(s, \mathit{successful}(\beta_i|^{o_i}))$ and this implies $(\cdot, r_i) = F(s_0, \mathit{successful}(\alpha_i \circ \beta_i|^{o_i}))$ by the definition of F. Thus, all operations in β_i satisfy the specification G as well. □

Lemma 9. *If some client C_j confirms an operation o_1 before an operation o_2, then o_2 does not precede o_1 in the execution history σ.*

Proof. Let μ_j denote the sequence of operations that C_j has confirmed up to o_2. According to the protocol logic (L40–L47), μ_j contains o_1, and o_1 has a smaller sequence number than o_2. Suppose o_2 was executed by C_i. Lemma 7 shows that μ_j is equal to the promised view to C_i of o_2, hence, o_1 is contained in the promised view to C_i of o_2. If C_i has confirmed o_1 earlier, then o_2 does not precede o_1. If o_1 is pending for o_2, then o_1 has been invoked by a client before o_2, as validated by C_i through verifying the corresponding INVOKE signature (L9). Since this occurs before o_2 completes, o_1 has been invoked before o_2 completed. □

Lemma 10. *The sequence π_i preserves the real-time order of σ.*

Proof. Recall that $\pi_i = \alpha_i \circ \beta_i$ and consider first the operations in α_i, which have been confirmed by some client. Lemma 9 shows that these operations preserve the real-time order of σ. Second, the operations in β_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, β_i also preserves the real-time order of σ.

We are left to show that no operation in β_i precedes an operation from α_i in σ. Recall that α_i consists of operations that have been confirmed and β_i are operations executed and committed by C_i. Let $\tilde{o}$ be the last operation of α_i and suppose it has sequence number l.

If some $C_k \neq C_i$ has confirmed $\tilde{o}$, then $\tilde{o}$ has been executed by C_i according to the definition of α_i, and $\tilde{o}$ has already completed before C_i invokes the first operation of β_i (which have sequence numbers larger than l), according to the assumption that σ is well-formed.

Otherwise, C_i has confirmed $\tilde{o}$ and some $C_j \neq C_i$ has executed $\tilde{o}$. Consider the time when C_i confirms $\tilde{o}$: C_j must have already completed $\tilde{o}$ because C_i verified the COMMIT signature for $\tilde{o}$ issued by C_j when $\tilde{o}$ completed. Since C_i has verified this signature for confirming $\tilde{o}$ (L41), C_i's local sequence-number variable c is at least l at that time. As operations of β_i have larger sequence numbers than l by definition, the protocol for handling REPLY messages (L11–L39) implies that all those operations were invoked after $\tilde{o}$ completed. □

Lemma 11. *If some operation $o_j \in \pi_i$ executed by C_j has been confirmed by a client C_k (including C_i), then $o_j \in \alpha_i$ and $\alpha_i|^{o_j} = \pi_i|^{o_j}$; furthermore, $\alpha_i|^{o_j}$ is equal to the promised view to C_j of o_j.*

Proof. Consider the case that $C_k = C_i$ has confirmed o_j. Then $o_j \in \alpha_i$ according to the definition of α_i. The second statement is an immediate consequence of Lemma 7, since C_i has confirmed o_j.

Otherwise, some $C_k \neq C_i$ has confirmed o_j. If $C_j = C_i$, then $o_j \in \alpha_i$ by definition since o_j has been confirmed. If $C_j \neq C_i$, then $o_j \in \alpha_i$ because $o_j \in \pi_i$ but $\beta_i = \pi_i \setminus \alpha_i$ contains only operations executed by C_i. The second statement is an immediate consequence of Lemma 7, since C_k has confirmed o_j and α_i is defined accordingly. □

Lemma 12. *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 implies that o has been confirmed. Lemma 11 implies that $\alpha_i|^o = \alpha_j|^o$.
2. $o \in \beta_i$ and $o \in \alpha_j$: This case cannot occur, since no client has confirmed operations from β_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 occur, since β_i and β_j contain only pending-self operations of C_i and C_j, correspondingly.

□

5 Conclusion

This paper has presented COP, the Conflict-free Operation verification Protocol, which lets a group of clients execute a generic service coordinated by a remote, but 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 non-conflicting operation sequences (but may sometimes abort conflicting operations);

In COP every client executes all operations of the common service and maintains the state, similar to a replicated state machine [23]. It is possible to improve the efficiency of COP for specific services that permit efficient authentication of remote state, in order to reduce the work of the clients and to keep the (potentially large) state only at the server. This goal can typically be achieved for functionalities that support authenticated data structures [28]. In a successor to this work, Brandenburger et al. [3] show how apply this method to protect the integrity and consistency of data in a cloud object store.

Efficient authenticated data structures are only available for certain functionalities. Therefore, an important direction for future work lies in combining generic protocols for cryptographically verifiable computation [29] with COP, to reduce the client workload for arbitrary computations and to guarantee integrity and consistency with fork-linearizability semantics to multiple clients.

Acknowledgments

We thank Marcus Brandenburger for interesting discussions and valuable comments. We are grateful to the anonymous reviewers for important and constructive 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; in part by the European Commission through the Horizon 2020 Framework Programme (H2020-ICT-2014-1) under grant agreements number 644371 WITDOM and 644579 ESCUDO-CLOUD; and in part by the Swiss State Secretariat for Education, Research and Innovation (SERI) under contracts number 15.0098 and 15.0087.

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] M. Brandenburger, C. Cachin, and N. Knežević. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. In D. Naor, G. Heiser, and I. Keidar, editors, *Proc. 8th ACM International Systems and Storage Conference (SYSTOR)*, May 2015.

[3] M. Brandenburger, C. Cachin, and N. Knežević. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. *ACM Transactions on Privacy and Security*, 20(3):8:1–8:30, Aug. 2017.

[4] E. Brewer. Towards robust distributed systems (invited talk). In *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, 2000.

[5] C. Cachin. Integrity and consistency for untrusted services. In I. Cerná et al., editors, *Proc. 37th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2011)*, volume 6543 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2011.

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

[7] C. Cachin, I. Keidar, and A. Shraer. Fail-aware untrusted storage. *SIAM Journal on Computing*, 40(2):493–533, Apr. 2011. Preliminary version appears in *Proc. DSN 2009*.

[8] C. Cachin and O. Ohrimenko. Verifying the consistency of remote untrusted services with commutative operations. In M. K. Aguilera, L. Querzoni, and M. Shapiro, editors, *Proc. 18th Conference on Principles of Distributed Systems (OPODIS)*, volume 8878 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2014.

[9] 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)*, pages 129–138, 2007.

[10] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with Frientegrity: Privacy and integrity with an untrusted provider. In *Proc. 21st USENIX Security Symposium*, 2012.

[11] 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.

[12] 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.

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[14] 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.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[16] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[17] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th Symp. Operating Systems Design and Implementation (OSDI)*, pages 121–136, 2004.

[18] 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.

[19] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

[20] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4), 2011.

[21] M. Majuntke, D. Dobre, M. Serafini, and N. Suri. Abortable fork-linearizable storage. In T. F. Abdelzaher, M. Raynal, and N. Santoro, editors, *Proc. 13th Conference on Principles of Distributed Systems (OPODIS)*, volume 5923 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2009.

[22] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. 21st ACM Symposium on Principles of Distributed Computing (PODC)*, 2002.

[23] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[24] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Proc. 13th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400, 2011.

[25] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.

[26] 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.

[27] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[28] R. Tamassia. Authenticated data structures. In G. Di Battista and U. Zwick, editors, *Proc. 11th European Symposium on Algorithms (ESA)*, volume 2832 of *Lecture Notes in Computer Science*, pages 2–5. Springer, 2003.

[29] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them. *Commun. ACM*, 58(2), Feb. 2015.

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

[31] 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.