Integrity and Consistency for Untrusted Services
(Extended Abstract)

Christian Cachin

IBM Research - Zurich
CH-8803 Rüschlikon, Switzerland
cca@zurich.ibm.com

Abstract. A group of mutually trusting clients outsources an arbitrary computation service to a remote provider, 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 integrity of the stored data, the correctness of the remote computation process, and the consistency of the provider's responses.

We present a novel protocol that guarantees atomic operations to all clients when the provider is correct and fork-linearizable semantics when it is faulty; this means that all clients which observe each other's operations are consistent, in the sense that their own operations, plus those operations whose effects they see, have occurred atomically in same sequence. This protocol generalizes previous approaches that provided such guarantees only for outsourced storage services.

Keywords: cloud computing, fork-linearizability, data integrity, computation integrity, authenticated data structure, Byzantine emulation.

1 Introduction

Today many users outsource generic computing services to large-scale remote service providers and no longer run them locally. Commonly called the *cloud computing model*, this approach carries inherent risks concerning data security and service integrity.

Whereas data can be stored confidentially by encrypting it, ensuring the *integrity* of remote data and outsourced computations is a much harder problem. 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. Such deviations from a correct specification can be very difficult to spot manually.

Suppose a group of clients, whose members trust each other, relies on an untrusted remote server for a collaboration task. For instance, the group stores its project data on a cloud service and accesses it for coordination and document exchange. Although the server is usually correct and responds properly, it might become corrupted some day and respond wrongly. This work aims at discovering such misbehavior, in order for the clients to take some compensation action.

When the service provides data storage (read and write operations only), some well-known methods guarantee data integrity. With only one client, a *memory checker* [1] ensures that a read operation always returns the most recently written value. If multiple clients access the remote storage, they can combine a memory checker with an external trusted infrastructure (like a directory service or a key manager in a cryptographic file system), and achieve the same guarantees for many clients.

But in the asynchronous network model without client-to-client communication considered here, nothing prevents the server from mounting a *forking attack*, whereby it simply omits the operations of one client in its responses to other clients. Mazières and Shasha [15] put forward the notion of *fork-linearizability*, which captures the optimal achievable consistency guarantee in this setting. It ensures that whenever the server's responses to a client A have ignored a write operation executed by a client B, then A can never again read a value written by B afterwards and vice versa. With this notion, the clients detect server misbehavior from a single inconsistent operation — this is much easier than comparing the effects of *all* past operations one-by-one.

This paper makes the first step toward ensuring integrity and consistency for *arbitrary computing services* running on an untrusted server. It does so by extending untrusted storage protocols providing fork-linearizability to a generic service protocol with fork-linearizable semantics. Previous work in this model only addressed integrity for a storage service, but could not check the consistency of more general computations by the server.

Similar to the case of a storage service, the server can readily mount a forking attack by splitting the group of clients into subgroups and responding consistently within each subgroup, but not making operations from one subgroup visible to others. Because the protocol presented here ensures fork-linearizability, however, such violations become easy to discover. The method therefore protects the integrity of arbitrary services in an end-to-end way, as opposed to existing techniques that aim at ensuring the integrity of a computing platform (e.g., the *trusted computing* paradigm).

Our approach requires that (at least part of) the service implementation is known to the clients, because they need to double-check crucial steps of an algorithm locally. In this sense, the notion of fork-linearizable service integrity, as considered here, means that the clients have collaboratively verified every single operation of the service. This strictly generalizes the established notion of fork-linearizable storage integrity. A related notion for databases is ensured by the Blind Stone Tablet protocol [20].

1.1 Contributions

We present the first precise model for a group of mutually trusting clients to execute an *arbitrary service* on an untrusted server S, with the following characteristics. It guarantees *atomic operations* to all clients when S is correct and *fork-linearizability* when S is faulty; this means that all clients which observe each other's operations are *consistent*, in the sense that their own operations, plus those operations whose effects they see, have occurred atomically in same sequence.

Furthermore, we generalize the concept of *authenticated data structures* [16] toward executing arbitrary services in an authenticated manner with multiple clients. We present a protocol for consistent service execution on an untrusted server, which adds $O(n)$ communication overhead for a group of n clients; it generalizes existing protocols that have addressed only the special case of storage on an untrusted server.

1.2 Related Work

Ensuring integrity and consistency for services outsourced to third parties is a very important problem, particularly regarding security in cloud computing [8].

A common approach for tolerating faults, including adversarial actions by malicious, so-called Byzantine servers, relies on replication [5]. All such methods, however, break down as soon as a majority of servers becomes faulty. We are interested in consistency for only one server, which is potentially Byzantine.

Our approach directly builds on *authenticated data structures* [16, 14, 19]; they generalize Merkle hash trees for memory checking [1] to arbitrary search structures on general data sets. Authenticated data structures consist of communication-efficient methods for authenticating database queries answered by an untrusted provider. In contrast to our setting, the two- and three-party models of authenticated data structures allow only one client as a writer to modify the content. Our model allows any client to issue arbitrary operations, including updates.

Previous work on untrusted storage has addressed the multi-writer model. Mazières and Shasha [15] introduce untrusted storage protocols and the notion of fork-linearizability (under the name of *fork consistency*), and demonstrate them with the SUNDR storage system [12]. Subsequent work of Cachin et al. [4] improves the efficiency of untrusted storage protocols. A related work demonstrates how the operations of a revision control system can be mapped to an untrusted storage primitive, such that the resulting system protects integrity and consistency for revision control [2].

FAUST [3] and Venus [18] extend the model beyond the one considered here and let the clients occasionally exchange messages among themselves. This allows FAUST and Venus to obtain stronger semantics, in the sense that they eventually reach consistency (in the sense of linearizability) or detect server misbehavior. In our model without client-to-client communication, fork-linearizability, or one of the related "forking" consistency notions [3], is the best that can be achieved [15].

Several recent cloud-security mechanisms aim at a similar level of service consistency as guaranteed by our protocol. They include the Blind Stone Tablet [20] for consistent and private database execution using untrusted servers, the SPORC framework [9] for securing group collaboration tasks executed by untrusted servers, and the Depot [13] storage system.

Orthogonal approaches impose correct behavior on a remote service indirectly, for instance through accountability in a storage service [21] or distributed systems [10]. Yet other work relies on trusted hardware modules at all parties [6, 7].

1.3 Organization

Section 2 describes the model and recalls fork-linearizability and other consistency notions. In Section 3 the notion of *authenticated service execution* is introduced, which plays the main role for formalizing arbitrary services so that their responses can be verified. Section 4 presents the fork-linearizable service execution protocol. The detailed analysis and generalizations are omitted from this extended abstract.

2 System Model

2.1 System

We consider an asynchronous distributed system consisting of n clients $C_1, \ldots, C_n$ and a server S. Every client is connected to S through an asynchronous reliable channel that delivers messages in first-in/first-out (FIFO) order. The clients and the server together are called *parties*. A *protocol* P specifies the behaviors of all parties. An *execution* of P is a sequence of alternating states and state transitions, called *events*, which occur according to the specification of the system components.

All clients follow the protocol; in particular, they do not crash. Every client has some small local trusted memory, which serves to store keys and authentication values. The server might be faulty and deviate arbitrarily from the protocol; such behavior is also called *Byzantine*. A party that does not fail in an execution is *correct*.

2.2 Functionality

We consider a *deterministic state machine*, which is modeled by a *functionality* F as follows. It maintains a *state* $s \in \mathcal{S}$, repeatedly takes some *operation* $o \in \mathcal{O}$ as input (o may contain arguments), and outputs a *response* $r \in \mathcal{R}$ and a new state s'. The initial state is denoted by s_{F0}. Formally, a step of F is written as

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

Because operations are executed one after another, this gives the *sequential specification* of F. We discuss the concurrent invocation of multiple operations later.

We extend this notation for executing multiple operations $o_1, \ldots, o_m$ in sequence, starting from an initial state s_0, and write

$$(s', r) = F(s_0, [o_1, \ldots, o_m])$$

for $(s_i, r_i) = F(s_{i-1}, o_i)$ with $i = 1, \ldots, m$ and $(s', r) = (s_m, r_m)$.

We define the *space complexity* of F, denoted by *SPACE*$_F$, to be the number of bits required to store the largest of its states, i.e.,

$$\mathit{SPACE}_F = \max_{s \in \mathcal{S}} |s|.$$

The space complexity determines the amount of local storage necessary to execute F.

2.3 Operations and Histories

Our goal is to emulate F to the clients with the help of server S. The clients invoke the operations of F; every operation is 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 F occurring in σ. An operation is *complete* in a history if it has a matching response. For a sequence of events σ, *complete*(σ) is the maximal subsequence of σ consisting only of complete operations.

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 execution is *well-formed* if the sequence of events at each client consists of 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.

2.4 Consistency Conditions

We now describe the formal consistency notions required from an untrusted service, formulated in terms of the possible views of a client. A sequence of events π is called a *view* of a history σ at a client C_i w.r.t. a functionality F if σ can be extended (by appending zero or more responses) to a history σ' such that:

1. π is a sequential permutation of some subsequence of $complete(\sigma')$;
2. $\pi|_{C_i} = complete(\sigma')|_{C_i}$; and
3. π satisfies the sequential specification of F.

Intuitively, a view π of σ at C_i contains at least all those operations that either occur at C_i or are apparent from to C_i from its interaction with F.

One of the most important consistency conditions for concurrent operations is linearizability, which guarantees that all operations occur atomically.

Definition 1 (Linearizability [11]). *A history σ is* linearizable *w.r.t. a functionality F if there exists a sequence of events π such that:*

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

The notion of fork-linearizability [15] (originally called *fork consistency*) requires that when an operation is observed by multiple clients, the history of events occurring before the operation is the same. For instance, when a client reads a value written by another client from a storage service, the reader is assured to be consistent with the writer up to the write operation.

Definition 2 (Fork-linearizability). *A history σ is* fork-linearizable *w.r.t. a functionality F 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. F;*
2. *π_i preserves the real-time order of σ;*
3. *(*No-join*) For every client C_j and every operation $o \in \pi_i \cap \pi_j$, it holds that $\pi_i|^o = \pi_j|^o$.*

We now recall the concept of a *fork-linearizable Byzantine emulation* [4]. It summarizes the requirements put on our service emulation protocol, which runs between the clients and an untrusted server. This notion means that when the server is correct, the service should guarantee the standard notion of linearizability; otherwise, it should ensure fork-linearizability.

Definition 3 (Fork-linearizable Byzantine emulation). *A protocol P* emulates *a functionality F* on a Byzantine server S with fork-linearizability *whenever the following conditions hold:*

1. *If S is correct, the history of every fair and well-formed execution of P is linearizable w.r.t. F; and*
2. *The history of every fair and well-formed execution of P is fork-linearizable w.r.t. F.*

2.5 Cryptographic Primitives

Our implementation uses *hash functions*, *digital signatures*, and *symmetric-key encryption*. We model them as ideal functionalities here. But all notions can be made formal in the model of modern cryptography.

A hash function H maps a bit string x of arbitrary length to a short, unique representation of fixed length. It is assumed to be collision-free, that is, no party can produce two different inputs x and x' such that $H(x) = H(x')$.

A digital signature scheme provides two operations, *sign* and *verify*. The invocation of *sign* takes an index $i \in \{1, \ldots, n\}$ and a bit string m as parameters and returns a signature ϕ with the response. The *verify* operation takes the index i of a client, a string m, and a putative signature ϕ as parameters and returns a Boolean value $b \in \{\text{FALSE}, \text{TRUE}\}$ with the response. It satisfies that *verify*$(i, m, \phi) = \text{TRUE}$ for all i and m if and only if C_i has executed *sign*$(i, m) = \phi$ before. Only C_i may invoke *sign*$(i, \cdot)$ and S cannot invoke *sign*. Every party may invoke *verify*.

A symmetric encryption scheme consists of a key generation algorithm, an encryption algorithm *encrypt* and a decryption algorithm *decrypt*. Initially a trusted entity runs the key generator and obtains a key $k \in \mathcal{K}$. Algorithm *encrypt* takes k and a message m as inputs and returns a ciphertext c. Algorithm *decrypt* takes k and a ciphertext c as inputs and returns a message m. For any k and m, it is required that *decrypt*$(k,$ *encrypt*$(k, m)) = m$. Furthermore, any party that obtains $c =$ *encrypt*(k, m) but has no access to k obtains no useful information about m.

3 Service Execution and Authentication

This section first introduces a model for executing the service F on server S such that operations are invoked by the clients. The primary task of S is to maintain the global state s of F; we intend this model for coordination services, shared collaboration spaces, lightweight databases, storage applications and so on, with small computational expense for every operation, but high demand on maintaining a consistent state.

Given this setting, the clients could simply send their operations to S and, since F is deterministic, S could execute them and return the responses. But we are interested in a model where the clients execute the bulk of every operation, so as to reduce the load on S. This assumption also helps preparing the ground for authenticating the responses of S.

In the second part of this section, we introduce a model for *authenticating* the execution of a sequence of operations issued by a single client (imagine for a moment there is only one client; we extend this to multiple clients later). The client uses its local trusted memory to maintain some authentication data, from which it verifies the responses of F sent by S. This model closely resembles the established concept of authenticated data structures.

3.1 Separated Execution

We model the execution of operations of F in a *separated way*, such that the clients do most of the work. Not all functionalities encountered require that every operation accesses the complete state s. An operation o can be executed in a separated way when it uses only a part s_o of the *global state* s of the functionality; this part may depend on the operation. If o modifies the global state, then the separated execution will also generate an updated state s'_o, which must be reconciled with s to maintain the correct semantics of F.

More formally, we say a functionality F allows *separated execution* when there exist three deterministic algorithms $\mathit{extract}_F$, exec_F, and $\mathit{reconcile}_F$ as follows. Algorithm $\mathit{extract}_F$ produces a *partial state* s_o from a global state s and an operation o,

$$s_o \leftarrow \mathit{extract}_F(s, o);$$

algorithm exec_F executes o on the partial state s_o to produce a response r and a *partial updated state* s'_o,

$$(s'_o, r) \leftarrow \mathit{exec}_F(s_o, o);$$

finally, algorithm $\mathit{reconcile}_F$ takes s'_o and o, together with the old global state s and outputs the new global state

$$s \leftarrow \mathit{reconcile}_F(s, s'_o, o).$$

The algorithms satisfy that for any $s \in \mathcal{S}$ and $o \in \mathcal{O}$, and for any s', r with $(s', r) = F(s, o)$, there exists a partial state $s_o = \mathit{extract}_F(s, o)$ and a partial updated state s'_o such that

$$(s'_o, r) = \mathit{exec}_F(s_o, o) \ \wedge \ s' = \mathit{reconcile}_F(s, s'_o, o)$$

and

$$|s_o| \ll |s| \ \wedge \ |s'_o| \ll |s'|.$$

In other words, the algorithms for the separated execution of F produce the same response and new state as the original F, but there exist intermediate states for the operation (s_o and s'_o), which are much smaller than the full state(s). The latter requirement should be understood qualitatively and is not quantified; but it is crucial for enabling efficient separated execution between a client and a server.

The *communication complexity* of some F with separated execution measures the size of the messages that must be communicated for separated execution. It is denoted by COMM_F and defined as the number of bits required to store the largest partial state s_o, partial updated state s'_o, together with a description of the operation o itself, for executing any operation on any state. That is,

$$\begin{aligned}\mathit{COMM}_F \;=\; \max\bigl\{ |s_o| + |s'_o| + |o| \;\bigm|\\ s \in \mathcal{S}, o \in \mathcal{O}, s_o = \mathit{extract}_F(s,o), (s'_o, r) = \mathit{exec}_F(s_o, o)\bigr\}.\end{aligned}$$

3.2 Authenticated Separated Execution

When only a single client engages in separated execution of operations on the server, well-known methods allow the client to verify the correctness of the responses. These methods protect the client from a faulty server that tries to forge wrong responses. Known generally as *authenticated data structures* [16, 14], they apply to a broad class of information retrieval services, such as reading an item from a memory, hash tables, or search queries to a structured data type. Such service authentication schemes rely on a small *authenticator* value maintained by the client in its local trusted memory. The client can verify the response of an operation o in such a way that it recognizes when the response differs from the correct response r, resulting from applying o to the current state s of the service. That is, state s is obtained by applying all past operations of the client to F in order and the correct response is determined by $(s', r) = F(s, o)$. We model this concept as an extension of separated execution.

We say a functionality F allows *authenticated separated execution* when there exist three deterministic algorithms $\mathit{authextract}_F$, $\mathit{authexec}_F$, and $\mathit{authreconcile}_F$ as follows. Algorithm $\mathit{authextract}_F$ produces a *partial state* s_o from a global state s and an operation o,

$$s_o \;\leftarrow\; \mathit{authextract}_F(s, o).$$

The client maintains an *authenticator* denoted by a, which is initialized to a default value a_{F0}. Algorithm $\mathit{authexec}_F$ takes a, s_o, and o as inputs and produces an *updated authenticator* a', a *partial updated state* s'_o, and a response r. In the course of executing o, the algorithm also *verifies* its inputs with respect to a and may output the special symbol $\perp$ as response, indicating that the verification failed. In other words,

$$(a', s'_o, r) \;\leftarrow\; \mathit{authexec}_F(a, s_o, o),$$

with $r = \perp$ if and only if verification failed. Finally, algorithm $\mathit{authreconcile}_F$ takes s'_o and o, together with the old global state s and outputs the new global state

$$s \;\leftarrow\; \mathit{authreconcile}_F(s, s'_o, o).$$

Its role is exactly the same as in separated execution.

A *proper authenticated execution* of the operation sequence $o_1, \ldots, o_m$ proceeds as follows. Starting with the initial authenticator $a_0 = a_{F0}$ and state $s_0 = s_{F0}$, it computes

$$\begin{aligned}(s_i, r_i) &\leftarrow F(s_{i-1}, o_i)\\ s_{o_i} &\leftarrow \mathit{authextract}_F(s_i, o)\\ (a_i, s'_{o_i}, r_i) &\leftarrow \mathit{authexec}_F(a_{i-1}, s_{o_i}, o_i),\end{aligned}$$

for $i = 1, \ldots, m$ and outputs the triple (a_m, s_m, r_m) containing an authenticator a_m, state s_m, and response r_m.

Consider now the proper authenticated execution of an arbitrary operation sequence and the resulting authenticator a and state s. The following conditions must hold:

Correctness: For any $o \in \mathcal{O}$ and $(s', r) = F(s, o)$, there exist $s_o = \mathit{authextract}_F(s, o)$ and a', s'_o, and $r \neq \bot$ such that

$$(a', s'_o, r) = \mathit{authexec}_F(a, s_o, o) \ \wedge \ s' = \mathit{authreconcile}_F(s, s'_o, o).$$

and

$$|a'| \ll |s| \ \wedge \ |s_o| \ll |s| \ \wedge \ |s'_o| \ll |s'|.$$

Security: For any $o \in \mathcal{O}$ and any adversary that outputs some $\tilde{s}_o$, suppose that there exist a' and s'_o such that $(a', s'_o, \tilde{r}) = \mathit{authexec}_F(a, \tilde{s}_o, o)$ with $\tilde{r} \neq \bot$; then $\tilde{r} = r$.

The *correctness* property is simply reformulated from the unauthenticated scheme for separated execution. It states that for any authenticator and state s resulting from a proper authenticated execution, applying separated execution of o yields a response $r \neq \bot$ such that verification succeeds and, moreover, the resulting updated state s' together with r satisfies $(s', r) = F(s, o)$.

The *security* property considers a faulty S as an adversary, which tries to forge some partial state $\tilde{s}_o$ that causes the client to produce a wrong response $\tilde{r}$. But in an authenticated separated execution scheme, algorithm $\mathit{authexec}_F$ either outputs the correct response ($\tilde{r} = r$), or it recognizes the forgery and the verification fails ($\tilde{r} = \bot$).

The *communication complexity* of some F with authenticated separated execution is defined in the same way as for separated execution and measures how much data must be communicated between C and S.

The notion of authenticated data structures [14] differs from a service with authenticated separated execution in that the former does not contain a partial updated state and the reconciliation step. In fact, the server could equally well execute the whole operation on the state that it maintains. But in practice, many algorithms execute update operations more efficiently when the client computes the updated parts of the state and the server merely stores them in its memory.

3.3 Examples

The literature contains many examples of data structures that can be formulated as functionalities with authenticated separated execution. They are interesting because their communication complexity for separated execution is much smaller their space complexity. For instance, hash trees can be used to check the correctness of individual entries in a memory with N elements [1] with complexity $O(\log N)$, a generalization of hash trees can authenticate responses produced by any DAG-structured query evaluation algorithm with logarithmic overhead [14], and cryptographic methods based on accumulators can maintain authenticated hash tables with constant communication for query operations and sub-linear cost for updates [17].

As a concrete example, consider a functionality *MEM* whose state consists of N storage locations denoted by *MEM*$[1], \ldots,$ *MEM*$[N]$. *MEM* supports two operations:

$\mathit{read}(j)$, which returns $\mathit{MEM}[j]$, and $\mathit{write}((j, x))$, which assigns $\mathit{MEM}[j] \leftarrow x$ and returns nothing. Note that for $N = n$ and when C_i may only write to $\mathit{MEM}[i]$, we obtain the functionality that was considered in most previous work on untrusted storage (e.g., [4]).

A standard hash tree computed over $\mathit{MEM}[1], \ldots, \mathit{MEM}[N]$ gives an authenticated separated execution scheme, where the internal nodes of the tree are also stored in the state of *MEM*. The authenticator is the root node of the hash tree, which commits all entries in *MEM*. Algorithm $\mathit{authextract}_{\mathit{MEM}}$ for an operation that concerns entry j always returns the internal tree nodes along the path from the root to the leaf node j and all their siblings, which are needed for recomputing the root hash in order to authenticate leaf node j [1]. Verification succeeds if the recomputed root hash matches the authenticator. For a write operation, the nodes on the path from $\mathit{MEM}[j]$ to the root are updated and included in the partial updated state s'_o. The server extracts them from s'_o and stores them in the appropriate place during $\mathit{authreconcile}_{\mathit{MEM}}$.

The client must explicitly recompute the path in the hash tree also for write operations, in order to verify the sibling nodes along the path from the modified leaf node to the root; these nodes originate from the server and influence the computation of the new root hash. If they are not verified, they might lead to an invalid authenticator. Because the client computes these values anyway, they are contained in the partial updated state, and the server only needs to store them.

In this way, our notion of authenticated separated execution models closely what happens in practical hash tree implementations inside cryptographic storage systems; this is not possible with the notion of an authenticated data structure, where no reconciliation algorithm is foreseen.

4 Fork-Linearizable Execution Protocol

We now introduce a novel untrusted service execution protocol, which emulates an arbitrary F on a Byzantine server with fork-linearizability. The protocol combines elements from existing untrusted storage protocols with an authenticated separated execution scheme for F.

The protocol operates in lock-step mode, similar to the bare-bones storage protocol of SUNDR [15]. This means that the server serializes all operations and does not allow them to execute concurrently. Proceeding in lock-step is for illustration purposes only; extending it to concurrent operations is feasible and discussed at the end of this paper.

At a high level, the protocol operates like this. A client assigns a local *timestamp* to every one of its operations. Every client maintains a timestamp vector T in its trusted memory. At client C_i, entry $T[j]$ is equal to the timestamp of the most recently executed operation by C_j in some view of C_i. To begin executing an operation o, client C_i sends a SUBMIT message with o to S. A correct S responds to this SUBMIT message by invoking the authenticated separated execution scheme, and computes $s_o \leftarrow \mathit{authextract}_F(s, o)$ on the current state s.

In addition to s, the server maintains a timestamp vector V, an authenticator a, and a signature φ, which it received in a so-called COMMIT message from the client C_c that executed the last preceding operation at S. The signature was issued by C_c on V and a. The server sends a REPLY message to C_i containing V, a, s_o, c, and φ.

When it receives the REPLY message, the client first checks the content. It verifies the signature φ and makes sure that $V \geq T$ (using vector comparison) and that $V[i] = T[i]$. If not, the client aborts the operation and halts, because this means that S has violated the consistency of the service.

Then C_i verifies the response with respect to a and runs the separated execution by computing $(a', s'_o, r) \leftarrow \mathit{authexec}_F(a, s_o, o)$. If the verification fails, the client again halts. Otherwise, C_i proceeds to copying the received timestamp vector V into its variable T, incrementing $T[i]$, and computing a signature φ' on T and a'. The value $T[i]$ becomes the *timestamp* of o. Finally, C_i returns a COMMIT message to S containing T, a', s'_o, and φ'.

It is not hard to see that all checks are satisfied when S is correct because every client only increments its own entry in a timestamp vector. Therefore, the timestamp vectors sent out by S in REPLY messages appear in strictly increasing order.

The description so far allows the server to learn the authenticator values, which is not foreseen in the model of an authenticated separated execution scheme. To prevent any damage that might be caused by this, all clients know a common secret key k for a symmetric encryption scheme and use it to encrypt the authenticator before sending it to S.

This completes the high-level description of the untrusted service execution protocol; the details are given in Algorithms 1 and 2.

Algorithm 1. Untrusted execution protocol for client C_i

```
State
    $k \in \mathcal{K}$                                        // symmetric encryption key
    $T \in \mathbb{N}_0{}^n$, initially $[0]^n$                // current timestamp vector

upon operation $\mathit{run}_F(o)$ do
    send message $[\textsc{submit}, o]$ to $S$
    wait for message $[\textsc{reply}, V, \bar{a}, s_o, c, \varphi]$
    if $\big(V = [0]^n \vee \mathit{verify}(c, \textsc{commit}\|V\|\bar{a}, \varphi)\big) \wedge V \geq T \wedge V[i] = T[i]$ then
        if $V = [0]^n$ then
            $a \leftarrow a_{F0}$
        else
            $a \leftarrow \mathit{decrypt}(k, \bar{a})$
        $(a', s'_o, r) \leftarrow \mathit{authexec}_F(a, s_o, o)$
        if $r \neq \bot$ then
            $T \leftarrow V$
            $T[i] \leftarrow T[i] + 1$
            $\varphi' \leftarrow \mathit{sign}(i, \textsc{commit}\|T\|a')$
            $\bar{a}' \leftarrow \mathit{encrypt}(k, a')$
            send message $[\textsc{commit}, o, T, \bar{a}', s'_o, \varphi']$ to $S$
            return $r$
    halt
```

Algorithm 2. Untrusted execution protocol for server S

```
State
    $s \in \mathcal{S}$, initially $s_{F0}$                                  // state of $F$
    $c \in \{1, \dots, n\}$, initially 1          // index of currently or most recently served client
    $V \in \mathbb{N}_0{}^n$, initially $[0]^n$          // timestamp vector of last committed operation
    $\bar{a}$, initially $\epsilon$                  // encrypted authenticator of last committed operation
    $\varphi$, initially $\epsilon$                           // signature of last committed operation
    $block \in \{\text{FALSE}, \text{TRUE}\}$, initially FALSE

upon receiving message $[\text{SUBMIT}, o]$ from $C_i$ such that $block = \text{FALSE}$ do
    $s_o \leftarrow authextract_F(s, o)$
    send message $[\text{REPLY}, V, \bar{a}, s_o, c, \varphi]$ to $C_i$
    $c \leftarrow i$
    $block \leftarrow \text{TRUE}$

upon receiving message $[\text{COMMIT}, o, T, \bar{a}', s'_o, \varphi']$ from $C_i$ such that $block = \text{TRUE} \wedge i = c$ do
    $s \leftarrow authreconcile_F(s, s'_o, o)$
    $(V, \bar{a}, \varphi) \leftarrow (T, \bar{a}', \varphi')$
    $block \leftarrow \text{FALSE}$
```

Intuitively, the algorithm relies on the same properties of vector clocks as previous protocols for untrusted storage [15, 4]. Note that S can only send a timestamp vector and authenticator in a REPLY message that have been signed by a client; otherwise, the first verification step in Algorithm 1 fails. Under this condition, S may violate the protocol only by sending a timestamp vector/authenticator pair that is properly signed but does not satisfy a global sequential order of the operations.

In other words, a violation by S means that there is one operation o_0 whose timestamp vector is received in a REPLY by at least two different clients C_1 and C_2, in operations o_1 and o_2, respectively. If all other information is correct, operations o_1 and o_2 both succeed, but the two clients sign *incomparable* timestamp vectors. According to the protocol, one can then show that C_1 will not execute any operation in a view at C_1 that includes o_2 and, vice versa, any operation in a view at C_2 that includes o_1 will cause C_2 to abort.

With the functionality *MEM* from the previous section and n storage locations, this protocol gives the same guarantees as the bare-bones storage protocol of SUNDR [15] and the lock-step protocol of Cachin et al. [4]. As in the latter protocol, our algorithm adds a linear (in n) overhead to the communication complexity of separated execution.

5 Conclusion

This paper has introduced the first precise model for a group of mutually trusting clients to execute an arbitrary service on an untrusted server S, such that the clients observe atomic operations when S is correct and the service respects fork-linearizability when S is Byzantine. An implementation of this notion has been obtained by combining any scheme for authenticated separated execution with elements from untrusted storage protocols.

The protocol is not particularly efficient because a correct server executes all operations in lock-step mode. Similar to untrusted storage protocols, the protocol can be improved by letting the clients execute some operations concurrently, as long as they do not conflict. Some restrictions on the achievable parallelism have been identified [4]. Clarifying the concept of conflicts for arbitrary functionalities and extending the protocol to concurrent operations are deferred to the forthcoming full version of this paper.

Acknowledgments

I thank Alexander Shraer, Idit Keidar, Rüdiger Kapitza, and Matthias Schunter for interesting discussions about the ideas leading to this paper.

This work has been supported in part by the European Commission through the ICT programme under contracts ICT-2007-216676 ECRYPT II and ICT-2009-257243 TCLOUDS.

References

[1] Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica 12, 225–244 (1994)

[2] Cachin, C., Geisler, M.: Integrity protection for revision control. In: Abdalla, M., Pointcheval, D. (eds.) ACNS 2009. LNCS, vol. 5536, pp. 382–399. Springer, Heidelberg (2009)

[3] Cachin, C., Keidar, I., Shraer, A.: Fail-aware untrusted storage. In: Proc. International Conference on Dependable Systems and Networks (DSN-DCCS), pp. 494–503 (2009)

[4] 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)

[5] Charron-Bost, B., Pedone, F., Schiper, A. (eds.): Replication: Theory and Practice. LNCS, vol. 5959. Springer, Heidelberg (2010)

[6] Chun, B.G., Maniatis, P., Shenker, S., Kubiatowicz, J.: Attested append-only memory: Making adversaries stick to their word. In: Proc. 21st ACM Symposium on Operating System Principles (SOSP), pp. 189–204 (2007)

[7] Chun, B.G., Maniatis, P., Shenker, S., Kubiatowicz, J.: Tiered fault tolerance for long-term integrity. In: Proc. 7th USENIX Conference on File and Storage Technologies, FAST (2009)

[8] Cloud Security Alliance, CSA (2010), http://www.cloudsecurityalliance.org/

[9] 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)

[10] Haeberlen, A., Kouznetsov, P., Druschel, P.: PeerReview: Practical accountability for distributed systems. In: Proc. 21st ACM Symposium on Operating System Principles (SOSP), pp. 175–188 (2007)

[11] 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)

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

[13] 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)

[14] 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)

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

[16] Naor, M., Nissim, K.: Certificate revocation and certificate update. IEEE Journal on Selected Areas in Communications 18(4), 561–570 (2000)

[17] Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables. In: Proc. 15th ACM Conference on Computer and Communications Security, CCS (2008)

[18] 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, New York (2010)

[19] Tamassia, R., Triandopoulos, N.: Computational bounds on hierarchical data processing with applications to information security. In: Caires, L., et al. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 153–165. Springer, Heidelberg (2005)

[20] 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)

[21] Yumerefendi, A.R., Chase, J.S.: Strong accountability for network storage. ACM Transactions on Storage 3(3) (2007)