Pipeline-Integrity: Scaling the Use of Authenticated Data Structures up to the Cloud

Diego Pennino\textsuperscript{a}, Maurizio Pizzonia\textsuperscript{b}, Federico Griscioli\textsuperscript{c}

\textit{Università degli Studi Roma Tre, Dipartimento di Ingegneria, Sezione Informatica e Automazione}
Via della Vasca Navale 79, 00146, Roma (Italy)
\textsuperscript{a}pennino@ing.uniroma3.it, \textsuperscript{b}pizzonia@ing.uniroma3.it, \textsuperscript{c}griscioli@ing.uniroma3.it

Abstract

Public cloud storage services are widely adopted for their scalability and low cost. However, delegating the management of the storage has serious implications from the security point of view. We focus on integrity verification of query results based on the use of Authenticated Data Structures (ADS). An ADS enables efficient updates of a cryptographic digest, when data changes, and efficient query verification against this digest. Since, the digest can be updated (and usually signed) exclusively with the intervention of a trusted party, the adoption of this approach is source of a serious performance degradation, in particular when the trusted party is far from the server that stores the ADS.

In this paper, we show a protocol for a key-value storage service that provides ADS-enabled integrity-protected queries and updates without impairing scalability, even in the presence of large network latencies between trusted clients and an untrusted server. Our solution complies with the principle of the cloud paradigm in which services should be able to arbitrarily scale with respect to number of clients, requests rates, and data size keeping response time limited. We formally prove that our approach is able to detect server misbehaviour in a setting whose consistency rules are only slightly

\textsuperscript{\#}This is the accepted manuscript of the corresponding article published in Future Generation Computing Systems. The associated formal publication can be obtained from Elsevier at https://doi.org/10.1016/j.future.2019.05.018. © 2019. This manuscript version is made available under the CC-BY-NC-ND 4.0 license http://creativecommons.org/licenses/by-nc-nd/4.0.
weaker than those guaranteed by previous results. We provide experimental evidence for the feasibility and scalability of our approach.

**Keywords:** Authenticated Data Structures; Scalability; Cloud; Integrity; Pipeline; Fork-Linearisability.

### 1. Introduction

Public cloud infrastructures are popular since they enable virtually unlimited scaling paying only the amount of needed resources, on-demand. However, the public cloud model, inherently implies that the user delegates to the cloud provider the management of the infrastructure, and, with it, many security guarantees that were clearly under her/his control with in-house solutions. Most users are concerned with data confidentiality, for which there are a large number of effective cryptography-based tools that can be used. These tools allow the user to keep control over data confidentiality while keeping the advantages of the public cloud. The same objective is still difficult to meet for “fully fledged” data integrity. For **data integrity** we intend the capability to detect (malicious or accidental) changes of user data that do not conform to the will of the user. Usually this capability is provided by (a trusted part of) the same system that manages user data. If there is only one user, just a sequence of updates is enough to specify the intention of the user. However, when many users concurrently update the data, additionally **consistency rules** should be specified (see Sections 4.3 and 4.4).

Integrity is usually taken for granted by users, but a cloud provider might change user’s data either by mistake or maliciously. For example, files or records might be changed, got lost, reverted to a previous version, or deleted files or records might be restored. Depending on the kind of application, injecting an outdated version of the data into a business process might lead to huge loss or damage. Typical sectors in which integrity is paramount are banking, financial, health, legal, defence, industrial control systems and critical infrastructures. For this reason, practical means to check the integrity of data returned by a public cloud before use (i.e., **on-line**) is highly desirable in these contexts. Further, both the user and the cloud provider might want to have a cryptographic proof of the genuineness or of the inaccuracy of the data to use for dispute resolution. Note that, for deletion, restoration, or reversion to previous version, signing each file or record individually does not help in detecting the tampering, while signing the whole dataset with conventional
techniques is effective but highly impractical for most applications. To solve this, authenticated data structures (ADS) \[38\] can be adopted. Using an ADS, an untrusted entity can store a dataset and can answer queries providing a proof of the validity of the answer to the client. Essentially, an ADS is a mean to keep a cryptographic hash of the dataset that enable efficient update of the hash upon small changes and efficient integrity checks of small parts of the dataset against a trusted version of that hash. Traditionally, ADSes are adopted in a model where a single source asks the untrusted entity (responder) to update the dataset and a plurality of users can perform authenticated queries to the responder. Users can validate the queries results against a cryptographic hash obtained from the source by a trusted channel. In another traditional model, a single client performs updates and queries to the untrusted storage.

A large body of research work deals with integrity of outsourced database with many different approaches that may favour security, efficiency, flexibility of the queries, etc. However, an on-line integrity verification system for a public cloud service need to fulfil very strict requirements to avoid impairing the advantages of the cloud adoption. In particular, the solution should scale with respect to the amount of data, updates, and clients, with the same approach the typical cloud storage solutions do. For this reason, the above mentioned idea of keeping up-to-date a cryptographic hash for a large dataset has been regarded by many authors as impractical (for example, see \[3, 33\]).

In this paper, we address the problem of adopting ADSes while maintaining the possibility to achieve high throughput keeping limited latency. We define throughput as the maximum rate of updates per second the system can process. We define latency (or response time) as the time elapsed from the update reception to the instant when the server is able to serve a read that includes that update. We use ADSes in a model that is different than the traditional ones. In our setting, a possibly large number of clients performs both authenticated updates and queries to a single untrusted storage. This model is quite challenging. Any change in the dataset is reflected in a change of the single hash of the whole dataset. This turns out to be a bottleneck, since for each change, the server should contact a trusted party for a signature. In existing proposals for the same setting, updates are usually applied in batches. However, processing of each batch starts after the completion of the previous one, since the signature of the previous hash is needed (see, for example, \[7\]). This is critical if the network latency, between the server and
the trusted party, is non-negligible. Additionally, clients may expect that deviations of the behaviour of the server from certain consistency rules to be detected. For example, the system should detect answers from the server that are not consistent with its past answers.

Our main contribution is a protocol, called pipeline-integrity protocol, that allows a server to ask (possibly far) trusted parties to authenticate the hash of a new version of a continuously updated dataset without hindering scalability of the whole system. We address the scalability problem by allowing the server to start independent authentication processes that can proceed pipelined. In this way, both network and trusted resources are shared among several concurrent authentications, achieving much higher resource usage. We also introduce a new concept of consistency in a security setting called quasi-fork-linearisability, which is compatible with our pipelining approach and is only slightly weaker than fork-linearisability [11]. Fork-linearisability is a form of strong consistency in which the server is allowed to partition clients so that a strongly consistent history (linearisable) is shown to each partition. This form of consistency is proven to be the best possible in a setting where clients cannot directly communicate.

We theoretically prove that the pipeline-integrity protocol allows us to achieve high throughput with practically bounded latencies, provides quasi-fork-linearisability, and enables clients to detect Byzantine servers that deviates from the required behaviour. We analytically and experimentally compare latency and throughput of the pipeline-integrity protocol against the traditional approach, which performs only one authentication at a time.

The rest of the paper is structured as follows. In Section 2, we review the state of the art. Section 3 is dedicated to background on authenticated data structures and their use. Section 4 introduces models and definitions that will be used in the rest of the paper. In Section 5, we analyse the performances of the typical interaction of ADS-based client-server protocols. In Section 6, we provide a brief and intuitive overview of the protocols described in the next three sections. In Section 7, we describe a simplified version of the pipelined-integrity protocol, which has weak consistency properties but clearly show the main ideas we propose to achieve scalability. In Section 8, we show a protocol, and a data structure, to achieve quasi-fork-linearisability that has poor scalability properties but turns out to be compatible with our pipeline-integrity technique. Section 9 describes the complete version of our pipelined-integrity protocol, which unifies the results of the two previous sections and provide both scalability and quasi-fork-linearisability. In Sec-
tion 10, we discuss how to cope with real (non ideal) communication and computation resources. Section 11 provides experimental evidence of the feasibility and scalability of our approach. Section 12 draws the conclusions.

2. State of the Art

A large wealth of research works has dealt with the problem to verify the correct behaviour of an untrusted storage and many of them explicitly refer to a cloud computing setting. A good survey of the research in this area can be found in [45].

The effectiveness of each approach can be evaluated with respect to several aspects. Some of the coordinates that are relevant for this paper are: (1) the presence of a trusted entity in the cloud or if only clients are trusted, (2) the number of clients supported, (3) the load of each client in terms of data stored and computation performed, (4) the efficiency of the client-server protocol, (5) the probability with which an anomalous behaviour of the server is detected, (6) the ability to deal with an unbounded number of queries, and (7) the support for efficient updates and the consistency model supported.

A proof of retrievability [21, 6, 36] is a compact proof by a filesystem (prover) to a client (verifier) that a target file is actually stored. The proof of data possession [4] adds the possibility of data recovery. A typical limitation of these schemes is that they can only be applied to a limited number of requests, decided upfront. Also, they usually do not support efficient update. Some works, such as [16, 24, 5], describe protocols that, up to a certain extent, admit the dynamic update of stored data. In [41], the proof of retrievability approach is enhanced so that updates are efficiently supported and a third party auditor can perform the verification.

Many of the works in this area adopt Authenticated Data Structures (ADS) [38], especially when dynamic data operations are required. ADSes have many advantages: they provide deterministic verification, support dynamic operations and require the clients to keep only a constant amount of data: a digest of the whole dataset. This approach has also the advantage to detect attacks like deletion or reversion to a previous (authentic) version of part of the data, which require to consider the dataset as a whole. ADSes were successfully adopted in many works concerning integrity of outsourced Databases (see, for example, [22, 46, 37, 44, 31]). A typical problem tackled in these works is to support a broad class of queries, efficiently. In research
about verifiable databases a randomized periodic verification process was proposed (see, for example, [13, 47]).

When using ADSes, the single digest must be updated and propagated to all clients at each update in a secure way. In fact, the approaches based on ADSes treat a dataset as a single object: if even only one bit is updated the whole dataset is considered updated. Some works explicitly rule out ADSes on the basis of their inefficiency in the client-server setting when high concurrency is needed [3, 33]. Proposing an efficient way of using ADSes in this setting is exactly the problem addressed by this paper. The work [3] proposes a system based on deferred verification which requires a trusted entity in the cloud (e.g., a special processors like Intel SGX Enclave [29]) which, however, works only in-memory. The work [33] proposes a dynamic solution based on signature aggregation [23], which are much more expensive than ADSes, in terms of cryptographic computation.

Whenever more clients can concurrently perform updates, a consistency problem arises. Consistency has a long-standing research history, which was developed mainly in the areas of databases and multiprocessors architectures with shared memory (see, for example, [42, 8, 40, 20]). Many papers address the problem of verifying the correct behaviour of an untrusted storage service in the context of concurrent accesses, with the focus of providing provable guarantees about consistency. A strong notion of consistency is embodied in the definition of linearisability [20], which essentially states two things.

1. The outcome of the operations on a shared object have to be consistent with a sequence of operations $H$ (a history) that conforms with the sequence of operations as invoked by each client. It should be noted that each client can only perform operations sequentially so, from the point of view of the client, they are totally ordered.

2. If two operations are not concurrently invoked (by distinct clients), they must appear in that order in $H$.

The history $H$ is “chosen” by the server or, more often, is the outcome of unpredictable network latencies. The server might maliciously show different values for $H$ to different clients. This violates the obvious notion of integrity in a shared environment. Ideally, we would like to impose that all clients see the same $H$. In [28], it is shown that this is impossible to achieve in a setting in which clients cannot directly communicate or are not synchronised. Hence, the authors introduce a weaker form of consistency.
called *fork-consistency*. This was lately renamed as *fork-linearisability* in [11]. Fork-linearisability admits that the clients observe an execution that may split into multiple linearisable “forks”, which must never join. In other words, the union of the $H$ shown to the clients must be a tree where, at a fork, the set of operations of the branches are pairwise disjoint. The security aspect in this definition is bound to the capability of the client to detect the fork as soon as the server tries to merge two forks, i.e., to propose to a client updates that were kept hidden to that client till that time. A system realising fork-linearisability is shown in [25] proposing a quite inefficient protocol. In [43, 15] ways to enforce fork-linearisability are proposed in a setting where the whole storage is replicate on each client. The research described in [25, 11, 9] allows to store the data on an (untrusted) server and use vector clocks to give to the clients a partial view of all operations executed on the data. It can be proven [11] that to ensure fork-linearisability a blocking condition is unavoidable. Namely, it is impossible to avoid situations in which a client must wait another client to perform some actions. The results in [43, 10, 7] show protocols that allow certain classes of operations to proceed without waiting. VICOS [7] is probably the work that is more akin to this paper, regarding targeted problems. It shows a protocol that allows several clients to share an ADS preserving fork-linearisability. This work does not address the problem of the throughput and put considerable burden on the clients, since each client have to process all updates on the data, even if they are performed by other clients.

The Depot storage system [26] provides *fork-causal* consistency, which is weaker than fork-linearisability but enables to join forked histories and to cope with eventual consistency. The Depot approach is compatible with typical availability and scalability requirements of the cloud but its form of consistency is harder to handle for applications.

In [10], a survey of works providing integrity in the dynamic client-server setting with different consistency guarantees is provided.

### 3. Background

In this section, we recall basic concepts, terminology and properties about *authenticated data structures (ADS)*, limiting the matter to what is strictly needed to understand the rest of this paper. Further details can be found in [38, 27].
For this paper, an ADS is a container of implicitly ordered key-value pairs, denoted \( \langle k, v \rangle \), which are also called elements. The content of the ADS at a given instant of time is its state. The ADS deterministically provides a constant-size digest of the set of the key-value pairs of its content with the same properties of a cryptographic hash of that set. We call it root-hash, denoted by \( r \). If any element of the set changes, \( r \) changes. It is hard to find a set of elements whose root-hash is a value given in advance. An ADS provides two operations, the authenticated query of a key \( k \) and the authenticated update of a key \( k \) with a new value \( v' \). A query returns the value \( v \) and a proof of the result with respect to the current value of \( r \). If a trusted entity safely stores the current \( r \), it can query the ADS and execute a check of the proof against its trusted version of \( r \) to verify that the query result matches what expected. The update operation on \( k \) changes \( v \) associated with \( k \) into a provided \( v' \) and changes \( r \) in \( r' \), as well. The interesting aspect is that a trusted entity that intends to update \( k \) can autonomously compute \( r' \) starting from the proof of \( \langle k, v \rangle \) that can be obtained by a query.

As an example, we briefly introduce a specific ADS, the Merkle Hash Tree (MHT), however, the same properties hold for others ADSs, like, for example, the authenticated skip list \[17\]. A MHT is a binary tree \( T \) that is composed of internal nodes and leaves, see Figure 1. Every leaf is associated with a key-value pair \( \langle k, v \rangle \). The tree is managed as a binary search tree. Let hash(\( \cdot \)) be a cryptographic hash function. Every node \( n \) is labelled by a cryptographic hash \( h(n) \). If \( n \) is a leaf, we define \( h(n) = \text{hash}(\langle k, v \rangle) \). If \( n \) is an internal node, with \( n' \) and \( n'' \) its children, \( h(n) = \text{hash}(h(n')|h(n'')) \). If \( n \) is the root

\[
\begin{align*}
0 &: h(0) = \text{hash}(\langle k_0, v_0 \rangle) \\
1 &: h(1) = \text{hash}(\langle k_1, v_1 \rangle) \\
2 &: h(2) = \text{hash}(\langle k_2, v_2 \rangle) \\
3 &: h(3) = \text{hash}(\langle k_3, v_3 \rangle) \\
4 &: h(4) = \text{hash}(h(0)|h(1)) \\
5 &: h(5) = \text{hash}(h(2)|h(3)) \\
6 &: h(6) = \text{hash}(h(4)|h(5))
\end{align*}
\]

Figure 1: An example of Merkle Hash Tree with four leaves and a binary structure. We evidenced the elements regarding the proof of \( \langle k_1, v_1 \rangle \).
of \( T \), \( r = h(n) \) is the root-hash of \( T \). Let \( k \) be a key in \( T \), and let \( l \) be its associated leaf. Consider the path \((n_1, n_2, \ldots, n_m)\), from \( l = n_1 \) to \( n_m \), where \( n_m \) is the root of \( T \). For each \( n_i \) \((i = 1, \ldots, m-1)\), let \( \tilde{n}_i \) be the sibling of \( n_i \). The proof for \( \langle k, v \rangle \) according to \( T \), denoted \( \text{proof}(T, \langle k, v \rangle) \) possibly omitting \( T \) and/or \( v \) for short, is the sequence \((h(\tilde{n}_1), d_1, h(\tilde{n}_2), d_2, \ldots, h(\tilde{n}_{m-1}), d_{m-1})\), where \( d_i \in \{L, R\} \) indicates if \( n_i \) is the left or the right child of \( n_{i+1} \). For example, according to Figure 1, \( \text{proof}(T, k_1) \) is the sequence \((h(0), R, h(5), L)\).

It is easy to see that, given \( \text{proof}(T, k) \) and \( \langle k, v \rangle \), it is possible to compute \( r \) and that creating a different proof that gives the same \( r \) implies braking hash(\( \cdot \)). Also, considering the update of \( \langle k, v \rangle \) into \( \langle k, v' \rangle \), the new root-hash \( r' \) can be easily computed from \( \text{proof}(T, k) \) just pretending that \( \langle k, v' \rangle \) is the value of \( l \). The proof that a key \( k \) is not present in \( T \) can be given by providing the \( \text{proof}(T, k_1) \) and \( \text{proof}(T, k_2) \), where \( k_1 \) and \( k_2 \) are two consecutive keys such that \( k_1 < k < k_2 \). After the authenticity of \( \text{proof}(T, k_1) \) and \( \text{proof}(T, k_2) \) is verified, the proof that \( k_1 \) and \( k_2 \) are consecutive can be obtained by checking that the sequences of \( d_i \) matches regular expressions \( R^*Lz \) for \( k_1 \) and \( L^*Rz \) for \( k_2 \), where \( z \) is a possibly empty common suffix. We do not go into the details of the addition and deletion of a key. We just note that incomplete binary trees can be allowed by minimal changes in the above definitions. Changing the structure of the tree, even without changing the dataset, changes the root-hash, so the tree structure is part of the state of the MHT, as well. This contradicts the hypothesis of the deterministic link between the root-hash and set of elements contained in the ADS. This is essentially a technical problem that can be solved, for example, by defining a canonical structure of the tree that deterministically depends on the contained elements and caring that every operation leaves the tree structure in the canonical state. Clearly, the trusted entity that is going to compute \( r' \) should get all needed information to re-create locally the correct path(s) from the leaves involved in the update to the root, exactly as the ADS is supposed to do. Another approach is to resort to associative cryptographic hash functions [30] so that root-hash is independent from the way leaves are grouped.

When we have a large set of elements stored in an ADS, but we only need authentication for a small number of them, known in advance, we can resort to the pruning technique. Pruning reduces the storage size of the tree, without changing the root-hash, by removing sections of the tree that are no longer needed for the expected queries. The basic idea is very simple. Whenever a
subtree have only unneeded leaves, we can remove all the subtree maintaining only its root. Pruning an ADS reduces the required space, preserves the root-hash, preserves the capability of producing proofs for the needed keys, and keeps security intact. Pruning is obvious for a MHT but also other ADSes may support it.

A typical example of use of an ADS is for outsourcing a key-value store in a single client setting, keeping in the client only the root-hash \( r \) while keeping the ADS in an untrusted server. In this setting the query and update operations are as follows.

**Query**\((k)\). Server returns \( v \) and \( p = \text{proof}(\langle k, v \rangle) \). The client verifies the consistency of \( v \) with \( p \) and the local copy of \( r \).

**Update**\((k, v')\). The client preventively performs Query\((k)\) getting \( v \) and \( p = \text{proof}(\langle k, v \rangle) \), and checks \( p \) against the local copy of \( r \). Then, the client pretends the stored value to be \( \langle k, v' \rangle \) and compute all values along the path of \( p \) accordingly. It comes up with a new value \( r' \) for the root-hash, which is considered the current root-hash for the next operation. Then, the client send the operation Update\((k, v')\) to the server and forget anything else but \( r' \). When the server receives Update\((k, v')\), it update the ADS accordingly recomputing all the hashes all the way up to the root. Its current root-hash should turn out to be exactly the \( r' \) computed by the client.

We say that a root-hash \( r \) contains an update \( u \) if \( u \) is part of the sequence of updates that was applied to the dataset before reaching the state corresponding to \( r \).

### 4. Models and Terminology

In this section, we provide basic definitions, assumptions and models we use throughout this paper. First, we introduce general assumptions and our definition of scalability. Then, we formally introduce a model of the service we intend to support assuming correct behaviour of all actors. Finally, we formally define the consistency model that will be supported by our approach in the case of a Byzantine server.
4.1. General Setting and Assumptions

The results of this paper are stated in the setting in which there are a (possibly large) number of mutually trusted clients, with limited storage that need to store and share an arbitrarily large amount of data. They do that by relying on an untrusted server. Certain special clients are in charge of authenticating operations invoked by regular clients. They do not invoke operations themselves. They are called authenticators and we reserve the word clients for regular clients. We collectively refer to clients and authenticators as trusted entities. In practice, if deemed convenient, one machine can play both roles. However, in this paper, we always deal with them as if they were separate entities.

Trusted entities can only communicate with the server and are not synchronised. For simplicity, we assume all network communications are reliable and timings predictable. In other words, we assume that no message is lost, no network congestion occurs, and the network behaves deterministically and consistently over time. Since for real systems this assumption does not hold in general, we discuss the issues arising when network and clients are not reliable and timings not predictable in Section 10. In that section, we also show how to deal with those issues.

Each trusted entity $e$ can sign data $d$, by asymmetric encryption. The signature is denoted by $[d]_e$. We also write $[d]$ when $e$ is not relevant. We assume that each trusted entity has certificates of all other trusted entities and hence can securely verify all signatures.

4.2. Scalability

For the purpose of this work, when we say that a service scales, we intend that it is possible to increase volume of operations, data size, and number of clients (by increasing hardware resources dedicated to the server or network bandwidth) while keeping the response time bounded. As we will see in the following, when ADSes are adopted, the client-server protocol plays a fundamental role in the scalability of the whole system. In particular, for all protocols described in this paper, part of the processing must be performed client-side or generally by a trusted entity. The usual approach blocks the server while waiting a reply from the trusted entity and shows very bad resource usage. In Section 5, we formally analyse a client-server protocol adopting this approach and we show that its response time very badly depends on the throughput. In Section 9, we propose a protocol that does not have this problem while keeping a strong notion of security.
4.3. Key-Value Stores and Consistency

We focus on key-value stores, where we assume keys and values have limited size. This assumption simplifies the description and the analysis of protocols and algorithms shown in this paper. In fact, under this assumption, the time taken to process or transmit each operation is bounded by a constant. This helps us in focusing the paper on the interesting aspects of our solution. In the rest of the paper, for simplicity, we assume the store offers only two kinds of operations: (1) \textit{read}, which gets the value \(v\) currently associated with a key \(k\) and (2) \textit{update} of a key \(k\) with a value \(v\), which creates \(k\) if it does not exist or delete it if \(v = \perp\). These basic functionalities are the core of the features provided by several commercial services and open-source projects of the NoSQL landscape, see for example [12, 2, 35].

Each operation begins with its \textit{invocation} at the client and terminates when its \textit{response} reaches the client. Invocation and response occur at certain instants of time and are called \textit{events}. A sequence of events is \textit{complete} if each invoke event is matched by one, and only one, following response event in the sequence, for the same operation, and vice versa. In the following, we mostly deal with complete sequences and omit to state it explicitly. Each operation spans an \textit{interval} of time between the sending of its invocation and the receiving of its response. Two operations are \textit{concurrent} if their intervals overlap otherwise one of the two \textit{precedes} the other and they are \textit{sequential}. A complete sequence of events is \textit{sequential} if all operations in it are pairwise sequential. In other words, in a sequential sequence of events, for each operation, its invocation event is followed by its response event with no other event in between. A sequential sequence also implies a total order on the operations of that sequence.

Often, we consider a sequential permutation \(\pi\) of a complete sequence of events \(\sigma\). This is essentially a way to represent a choice of an order of the operations cited in \(\sigma\). For example, let \(\sigma = i_1r_1i_4i_2i_3r_2r_3r_4\). Where \(i_x\) and \(r_x\) denote invocation and response events of operation \(x\). A possible sequential permutation of \(\sigma\) is \(\pi_1 = i_1r_1i_4r_4i_2r_2i_3r_3\), expressing the order of operations 1 \(\ldots\) 3. Another possible sequential permutation of \(\sigma\) is \(\pi_2 = i_3r_3i_1r_1i_2r_2i_4r_4\), expressing the order of operations 3 \(\ldots\) 4. We note that \(\pi_1\) respects the precedence of operations implied by \(\sigma\) while \(\pi_2\) does not (operations 1 and 3 are reversed).

\textit{Consistency} is the property of a distributed system to behave according to the expected semantic of the operations as in a sequential setting, at least up to a certain extent. Typically, consistency guarantees are formalized in a
Figure 2: Relationships between operations and commit phases. Operations are represented by horizontal bars, where invocation is received by the server at the left extreme of the bar while the operation is considered concluded by the server at the right extreme of the bar. Each update is labelled $u$ and always ends during its associated commit. Each read is labelled with $r$, cannot end during a commit, and it is associated with its preceding commit.

setting in which operations are partitioned in \textit{sessions}, where the operations of each session are sequential and hence fully ordered in time. Sessions are supposed to be associated with a client, which expects to see the sequential behaviour of the operations if no other client interferes. In our model, the interactions between clients and the server, deviates a bit from this approach. We now formally describe this interaction. Since, in our setting, consistency is tightly linked with security, the formal definition of our consistency model is provided in Section 4.4.

We allow each client to invoke operations concurrently. We force update operations to be executed only during \textit{commit} phases, which are periodically triggered. Updates are applied respecting the invocation order of each client, but updates invoked by distinct clients can be arbitrarily interlaced by the server. Reads can be executed at any time but not during a commit. They return values according to the state of the key-value store as updated by the preceding commit. Figure 2 pictorially shows an example of how read
and update operations can evolve over time. At the end of a commit, for each executed update, a corresponding response is sent to the invoking client. While for a practical implementation this response is optional, in our model we always consider it.

More formally, we denote by \( \sigma \) a real-time ordered sequence of events. The invocation event of operation \( o \) is denoted by \( \text{inv}(o) \), its response event by \( \text{res}(o) \). Operations in \( \sigma \) are possibly concurrent. We associate with events a server-time that, for invocation, is the arrival time at the server and, for responses, is the sending time from the server. For an operation \( o \), its server-time interval is denoted \( I_o = (t_{\text{inv}(o)}, t_{\text{res}(o)}) \).

A commit is an atomic procedure executed on the server in a time interval \((t_{\text{begin}}, t_{\text{end}})\) during which no other operation can change the state of the key-value store. For brevity, we may treat commits as intervals of time to simplify notation. Commits do not overlap. An update \( u \) is associated with commit \( \chi \), if \( t_{\text{res}(u)} \) is in \( \chi \) and \( u \) is executed in the context of \( \chi \). Each update is associated with one and only one commit. The only way to change the content of the key-value store is to commit updates. A read operation \( r \) is associated with a commit \( \chi \), if \( \chi \) is the last commit before \( t_{\text{res}(r)} \). If \( r \) is executed before all commits, it is associated with no commit, and it is called initial. Each non initial read is associated with one and only one commit. A read operation returns a result on the basis of the state of the key-value store after the associated commit or on the basis of the initial state of the store for initial read operations.

Consider a sequence \( \sigma \) of events. The server executes read and update operations according to a certain sequential permutation \( \pi \) and is supposed to apply updates only during commits. One may ask if \( \pi \) is consistent with the commits. The following definition formally describes this.

**Definition 1 (Commit-correctness).** A sequential permutation \( \pi \) of a complete sequence of events \( \sigma \) is commit-correct with respect to the sequence of commits \( \chi_1, \ldots, \chi_n \) if

\[
\pi = \rho_0 \omega_1 \rho_1 \ldots \omega_i \rho_i \omega_{i+1} \rho_{i+1} \ldots \omega_n \rho_n
\]

where

1. \( \rho_0 \) is a sequential permutation of all and only the events of the initial read operations in \( \sigma \),
2. \( \rho_j \), with \( j = 1, \ldots, n \), is an arbitrary sequential permutation (of events) of read operations associated with \( \chi_j \), and

3. \( \omega_j \), with \( j = 1, \ldots, n \), is an arbitrary sequential permutation (of events) of update operations associated with \( \chi_j \) that conforms to the invocation order of each client.

Two operations commute if they provide the same results and state changes independently on the order they are executed. In our case, any two read operations associated with the same commit always commute. In all other cases, this property depends on the keys involved and in general may not commute. This definition can be naturally extended to a set of operations. Reordering read operations associated with different commits is forbidden in our setting, so it does not make sense to ask if they commute.

In the following, we introduce consistency (see Definitions 2 and 5), where a role is played by preservation of real-time order of events when permuting them. The following lemma states the relation between commit-correctness and preservation of the real-time order.

**Lemma 1.** Given a complete sequence of events \( \sigma \) and a sequence of commits \( \chi_1, \ldots, \chi_n \) such that all and only update operations end during a commit, let \( \pi \) be one sequential permutations of \( \sigma \). If \( \pi \) is commit-correct with respect to \( \chi_1, \ldots, \chi_n \), it preserves the real-time order of all non commuting operations of \( \sigma \).

**Proof.** We prove the statement by induction on the number \( n \) of commits. In the base case, \( n = 0 \) and \( \pi = \rho_0 \) which only contains read operations. Since all operations in \( \pi \) commute the statement is trivially true. Now, we prove the inductive case. Suppose the statement is true for \( \pi' = \rho_0 \omega_1 \rho_1 \omega_2 \rho_2 \ldots \omega_{n-1} \rho_{n-1} \), we prove the statement is true for \( \pi = \pi' \omega_n \rho_n \).

Consider a non commuting pair of operations. If they are both in \( \pi' \), they are in real-time order by the inductive hypothesis. Now, we prove that any \( o \in \omega_n \rho_n \) is in real-time order with any distinct \( o' \), if they do not commute (i.e., if they are not both read operations).

If \( o \) is a read operation then \( o \in \rho_n \). Operation \( o \) can not occur in \( \sigma \) before an operation \( o' \in \pi' \omega_n \) because \( o \) is associated with commit \( \chi_n \) and must end after \( \chi_n \). Hence, if \( o' \) is an update \( u \) associated with \( \chi_j \), \( t_{\text{res}(u)} \in \chi_j \leq \chi_n < t_{\text{res}(o)} \) with \( j \leq n \) and, if \( o' \) is a read \( r \) associated with \( \chi_j \), \( \chi_j < t_{\text{res}(r)} < \chi_{j+1} \leq \chi_n < t_{\text{res}(o)} \) with \( j < n \). This means that \( o \) and \( o' \)
are either correctly ordered or concurrent. Clearly, if \( o' \) is in \( \rho_n \), too, they commute and the statement does not apply.

If \( o \) is an update operation then \( o \in \omega_n \). Following the same reasoning as above, if \( o' \) is an update \( u \) associated with \( \chi_j \), \( t_{\text{res}(u)} \in \chi_j \leq \chi_n \ni t_{\text{res}(o)} \) with \( j < n \). If \( j = n \), \( u \) and \( o \) are concurrent, hence, they are not real-time ordered. If \( o' \) is a read \( r \) associated with \( \chi_j \), \( \chi_j < t_{\text{res}(r)} < \chi_j + 1 \leq \chi_n \ni t_{\text{res}(o)} \) with \( j < n \). Again, this means that \( o \) and \( o' \) are either correctly ordered or concurrent.

4.4. Threat Model and Consistency

Clients rely on an untrusted service to store their data. We suppose the server is operated or hosted by a cloud provider, which may change the data stored in it, deliberately or by mistake. In this paper, we assume that all trusted entities (hence all clients) trust each others and the only possibly malicious actor is the server. A fundamental requirement of our approach is that it should allow the clients to recognize any data tampering, right after the reception of the data. We also mandate that this should be done with high probability, so that it can be considered deterministic for any practical purpose (like many cryptographic hash properties are). The attacker can either be the cloud operator itself or be a third party that compromises the server forcing it to behave maliciously. From our point of view, both situations are attacks that we aim to detect and we do not distinguish them in the rest of the paper.

To define clearly our threat model, i.e., to distinguish between honest and malicious behaviour, we formally define our consistency model. We first introduce some basic definitions. We consider a set of clients, denoted by \( C \), that ask the server to perform possibly concurrent operations (read or update). Consider the invoke and response events corresponding to these operations. Events occurring in the system are totally ordered in a sequence \( \sigma \), according to their (invocation sending or response reception) real-time instant at the client. A sub-sequence of \( \sigma \) is an ordered subset of \( \sigma \) whose order conform to that of \( \sigma \).

We can consider a subsequence \( \sigma_i \) of \( \sigma \), for each client \( c_i \in C \), so that (at least) all completed operations occurring at \( c_i \) are in \( \sigma_i \). It is also useful to consider a sequential permutation \( \pi_i \) of \( \sigma_i \), which is essentially a

\footnote{Actually, here and in the following definitions of fork-linearisability and quasi-fork-linearisability we might restrict \( \sigma_i \) to contain only completed operations occurring at \( c_i \).}
sequence of operations, expressing the order in which the effect of those operations should be considered when executed according to their sequential semantic specification. A specific kind of consistency is defined in terms of the existence of $\sigma_i$ and $\pi_i$ satisfying certain conditions. The following is the traditional definition of the fork-linearisability consistency adapted from [11], for our definition of key-value store.

**Definition 2 (Fork-Linearisability).** A sequence of events $\sigma$ is fork-linearisable with respect to the semantic of a key-value store, if and only if, for each client $c_i$, there exists a complete subsequence $\sigma_i$ of $\sigma$ and a sequential permutation $\pi_i$ of $\sigma_i$ such that

1. all completed operations of $\sigma$ occurring at client $c_i$ are in $\sigma_i$,
2. $\pi_i$ preserves the real-time order of $\sigma$,
3. the operations of $\pi_i$ satisfy the semantic of their sequential specification, and
4. for every $o \in \pi_i \cap \pi_j$, the sequence of the events that precede $o$ in $\pi_i$ is the same as the sequence of the events that precede $o$ in $\pi_j$.

Definition 2 should be intended in monotonic sense. That is, consider the instants in which clients receive operation responses, denoted by $t_1, t_2, \ldots$. Any consistency definition should hold for the sequences $\sigma_{ij}$ and $\pi_{ij}$ seen by each client $c_i$ at each instant $t_j$. Clearly, we expect from a system a consistent monotonic behaviour in the sense that, $\sigma_{ij}$ and $\pi_{ij}$ should be prefix of $\sigma_{ij}^{j+1}$ and $\pi_{ij}^{j+1}$ respectively. This aspect is largely left implicit in previous literature, however, in the following we provide definitions that explicitly take it into account.

Condition 2 of Definition 2 makes sense for the general case tackled by [11] (a generic functionality) but is unnecessarily restricting in our case (a key-value store). Consider two read operations $r_1, r_2$ appearing in this real-time order in $\sigma$. Suppose no update operation is between $r_1$ and $r_2$ or is concurrent to them in $\sigma$. Clearly, preserving their real-time order is irrelevant. In

---

This change would not affect the following theory. However, we decided to avoid unneeded changes, with respect to definitions that can be found in literature, in order to ease the comparison of results.
general, it is important to preserve the real-time order only for operations that do not commute. For this reason, our consistency definition, provided in the following, relaxes that condition.

Note that, Definition 2 does not refer to the fact that the operations invoked by each client should be sequential, hence, it applies also to our setting that do not force each client to invoke operations sequentially (see Section 4.3).

Condition 4 of Definition 2 embodies the possibility that, at a certain instant, the server can partition clients showing to two distinct partitions different histories ($\pi_i$) that “fork” starting from a certain common event. Fork-linearisability is a fork-allowing variant of the definition of linearisability [20], which is considered a strong form of consistency in the literature that assume the server is not Byzantine. However, authors of [30] prove that certain kind of Byzantine behaviour (the forks) cannot be detected. On the contrary, a suitable protocol can detect if the server deviates from fork-linearisability behaviour, where forks are accepted if they do not join again. In the following, we further slightly relax fork-linearisability to make it compatible with our pipelining approach.

**Definition 3.** Two sequences $\pi_1$ and $\pi_2$ are disjoint-forking iff $\pi_1 = \alpha\beta_1$, $\pi_2 = \alpha\beta_2$, with $\alpha$ maximal and non-empty, and either $\beta_1 = \beta_2 = \emptyset$ or $\beta_1 \cap \beta_2 = \emptyset$.

A set of $n$ pairwise disjoint-forking sequences constitutes a tree (a path with no fork is a special case) with at most $n$ leaves, and after each fork the two branches have to be set-disjoint.

The following property links Definition 3 with Condition 4 of Definition 2.

**Property 1.** Two sequences $\pi_1$ and $\pi_2$ are disjoint-forking if and only if for every $o \in \pi_1 \cap \pi_2$, the sequence of the events that precede $o$ in $\pi_1$ is the same as the sequence of the events that precede $o$ in $\pi_2$.

**Proof.** First, we prove the necessary condition. Let $\pi_1 = \alpha\beta_1$ and $\pi_2 = \alpha\beta_2$. In the case $\beta_1 = \beta_2 = \emptyset$ the proof is trivial since the thesis holds for all $o \in \pi_1 = \pi_2$. In the case $\beta_1 \cap \beta_2 = \emptyset$, we have $\pi_1 \cap \pi_2 = \alpha$, hence for every $o \in \pi_1 \cap \pi_2 = \alpha$, the preceding elements in $\pi_1$ and $\pi_2$ are the same, by construction of $\alpha$.

Now, we prove the sufficient condition. Consider the latest (i.e., right-most) $o$ for which it holds $o \in \pi_1 \cap \pi_2$. The preceding events are the same in
\( \pi_1 \) and \( \pi_2 \) by hypothesis. We denote \( \alpha \) this prefix, which contains \( o \), and the remaining parts \( \beta_1 \) and \( \beta_2 \), so that \( \pi_1 = \alpha \beta_1 \) and \( \pi_2 = \alpha \beta_2 \). Sequence \( \alpha \) is maximal by construction and non empty since contains \( o \), at least. If \( \pi_1 = \pi_2 \) then \( \beta_1 = \beta_2 = \emptyset \). If \( \pi_1 \neq \pi_2 \), \( \beta_1 \) and \( \beta_2 \) are not empty, but \( \beta_1 \cap \beta_2 = \emptyset \), otherwise \( o \) would not be the latest event satisfying \( o \in \pi_1 \cap \pi_2 \). 

Property \( \square \) justifies the introduction of a weaker form of disjoint-forking and the corresponding slightly weaker form of fork-linearisability, which are defined in the following and will be used in Section \( \S \).

**Definition 4 (Quasi-Disjoint-Forking).** Two sequences \( \pi_1 \) and \( \pi_2 \) are quasi-disjoint-forking iff \( \pi_1 = \alpha \beta_1 \), \( \pi_2 = \alpha \beta_2 \), with \( \alpha \) maximal and non-empty, and either \( \beta_1 = \beta_2 = \emptyset \) or the following holds. Let \( O^c \) be the operation invoked by client \( c \) in \( \beta_1 \cap \beta_2 \). For each client \( c \), all \( o \) in \( O^c \) are invoked before (in the real-time order) the first response to an invocation in \( \beta_1 \cup \beta_2 \).

The above definition is clearly weaker than Definition \( \exists \) allowing partial overlap of branches, however, it states that those overlaps are limited. The extent of this limit depends on when responses are received by \( c \). For example, \( c \) may will to wait a response for an operation \( o \) in order to be sure that the following updates are in the same branch of \( o \), in case of malicious server. Definition \( \square \) motivates the introduction of the following.

**Definition 5 (Quasi-Fork-Linearisability).** A sequence of events \( \sigma \) is quasi-fork-linearisable with respect to the semantic of a key-value store, if and only if for each client \( c_i \), there exists a complete subsequence \( \sigma_i \) of \( \sigma \) and a sequential permutation \( \pi_i \) of \( \sigma_i \) such that

1. all completed operations of \( \sigma \) occurring at client \( c_i \) are in \( \sigma_i \),
2. \( \pi_i \) preserves the real-time order of \( \sigma \) of all non commuting operations,
3. the operations of \( \pi_i \) satisfy the semantic of their sequential specification, and
4. each pair \( \pi_i, \pi_j \) is quasi-disjoint-forking.

We note that, Conditions \( \exists \) and \( \square \) of Definition \( \exists \) are slightly weaker forms of the ones that are present in Definition \( \exists \) while the other conditions are the same.

The capability of a protocol to detect deviation from quasi-fork-linearisability is formalised by the following definition adapted from \( \square \).
Definition 6 (Byzantine Emulation). A protocol $P$ emulates a key-value store on a Byzantine server with quasi-fork-linearisability, if in every admissible execution of $P$ the sequence of events observed by the clients is quasi-fork-linearisable in monotonic sense. Moreover, if the server is correct, then every admissible execution is complete and has a linearisable history.

It is worth to further elaborate our comments following Definition 4. Suppose that all clients always wait the response to the previous operation before invoking the new one. In this case, Definition 3 holds, hence, fork-linearisability is guaranteed. In this sense, quasi-fork-linearisability can be regarded as a generalisation of fork-linearisability, which allows one to trade consistency for efficiency (see Section 9).

5. The Blocking Approach

The aim of this section is to show formally how adopting ADSes in a client-server setting with a naive protocol falls short of scalability. Our analysis shows that the throughput of the system (i.e., the maximum update invocation rate the system can sustain) can be approximated only at the cost of a very high latency (i.e., the time an update takes to be included in a read).

In our setting, we have many trusted entities that share a single root-hash. Since they cannot communicate directly, the common way to share the root-hash is to sign it and store it in the server. Clearly, only a trusted entity can legitimately update it. When the dataset have to be updated, the server must ask a trusted entity, an authenticator, to perform due checks and sign the new root-hash. The authenticator performs the checks on the basis of proofs derived from the current instance of the ADS and possibly other information. The kind of checks the trusted entity performs before signing the root-hash are responsible of the level of consistency guarantees provided by the whole system.

We introduce a very simple protocol, which can be regarded as an abstraction of the authentication part of other protocols described in literature (for example, see [7, 14, 41, 34]). For the sake of simplicity, in this section, we focus on the interaction scheme among the actors, disregarding all security and consistency aspects that are not strictly needed. We call it blocking protocol, since its main characteristic is that while the server is waiting a signed root-hash from a trusted entity $c_1$, it cannot ask another trusted entity $c_2$ to
sign another root-hash. In fact, the checks that $c_2$ should perform are usually based on data that is part of the reply from $c_1$, for example the signature of the root-hash provided by $c_1$. We analyse the performance of this protocol in terms of the relation between throughput and response time.

The server keeps a dataset $D$ equipped with an ADS. A group of update operations are applied to $D$ during a commit as described in Section 4.3. After each commit $D$ changes version. We denote the versions of $D$ by $D_i$, where $i$ is the index of the version. Version $D_i$ has root-hash $r_i$. The authentication of $D_i$ is $[r_i]_a$, which means that trusted entity $a$ has checked that $D_i$ derives from $D_{i-1}$ by the application of a certain set of updates that conforms to certain consistency rules.

We consider three different roles.

**Client.** It is a trusted entity in charge of invoking operations.

**Server.** It is in charge of executing operations and sending the response to the client along with an authentication that the server should obtain from an authenticator.

**Authenticator.** It is a trusted entity in charge of providing the authentication for the next version of the dataset upon server request.
Figure 3 depicts an example of interaction according to the blocking protocol. A client starts an operation sending an update invocation or a read invocation to the server.

The read invocation specifies the key $k$ to read. The server gets the value $v$ associated with $k$ and generates the corresponding proof($k$) against the current root-hash authentication $[r_i]$. The read response, sent from the server to the client, contains $k$, $v$, proof($k$), and $[r_i]$. The client, at the receiving of the response, verifies the consistency of $k$, $v$ and proof($k$) against $[r_i]$.

The update invocation specifies the key $k$ to update and the new value $v'$. At its reception, the server can perform the update procedure autonomously but cannot produce the authentication for the new version of dataset, since it cannot sign the new root-hash. It sends an authentication request to an authenticator containing $k$, its current value $v$, proof($k$), $v'$, $[r_{i-1}]$, where $i-1$ is the current version index.

Upon reception of an authentication request, the authenticator performs the following actions.

1. It checks proof($k$) of $k$, $v$, against $[r_{i-1}]$.
2. It computes $r_i$ from $k$, $v'$, and proof($k$).
3. It sends the authentication reply to the server containing $[r_i]$.

To increase the throughput, we allow queuing several update invocations and let the server asks an authenticator to authenticate all of them, cumulatively.

When the server receives an authentication reply, it updates the value of $D$ from $D_{i-1}$ into $D_i$, by exploiting the same information that were present in the request, and consider $[r_i]$ as the current authentication. It also sends to all clients, whose updates were executed, an update response.

Now, we analyse the scalability of the blocking approach. We call authentication round (or simply round) the process that start when the server sends an authentication request and ends when it receives the authentication reply. In the blocking protocol, there is only one round ongoing at a time. We denote by $T$ the duration of a round. We denote by $\lambda$ the frequency according to which update requests are received by the server, expressed in update requests per unit of time. We assume $\lambda$, as well as other parameters, to be constant in time. Let $m = \lambda T$ be the number of update requests received by the server during a round. If $\lambda$ is big enough $m > 1$, hence, when
a round terminates, there are already further update requests queued. We assume the server immediately starts a new authentication round when the previous one ends. This setting is depicted in Figure 4. Let $t_S$ to be the time needed by the server to prepare one update to be sent to the authenticator. For simplicity, we assume that all update requests take the same time $t_S$, $m$ updates take time $mt_S$. Let $t_N$ be the time needed to put an update request into the network for transmission. For simplicity, we assume that all update requests take the same time $t_N$ and, if $m$ update requests are cumulated into one authentication request, they take time $mt_N$ to be transmitted. We denote by $d$ the transmission (one-way) delay of the network. We assume this delay to be symmetric. We assume no network errors. Let $t_A$ be the time taken by the authenticator to process one update request. For simplicity, we suppose that the processing time is the same for all updates and if $m$ update requests are cumulated into one authentication request, the time taken by the authenticator to process all of them is $mt_A$. We assume all other overheads to be negligible, as well as the transmission time of the authentication reply. It holds that

$$T = 2d + m(t_S + t_N + t_A).$$

(1)
Figure 5: In the blocking protocol, the duration of an authentication round ($T$) hyperbolically goes to infinite when the arrival frequency of update requests ($\lambda$) approaches the maximum throughput ($\tau$).

If we suppose the system to work at steady pace, we can substitute $m = \lambda T$, getting

$$T = \frac{2d}{1 - \lambda(t_S + t_N + t_A)}.$$  

(2)

Figure 5 shows how $T$ changes with $\lambda$ according to Equation 2. The maximum throughput is $\tau = \frac{1}{t_S + t_N + t_A}$. Since a client can see its updates requests accepted only after that the authenticator replies, $T$ is a lower bound of the response time and goes hyperbolically with $\lambda$.

We observe that resources tend to be mostly idle. For simplicity, we suppose $t = t_S = t_N = t_A$. The fraction of the round for which each resource is busy is $mt/T = \lambda T t / T = \lambda t$, hence, the idle time ratio for each resource is $1 - \lambda t$. Note that, decreasing $t$ (i.e., increasing the speed of the resources) so that $T$ approaches $2d$, makes the idle time ratio to approach 1.

Clearly, increasing the throughput $1/t$ of the resources increases the cost of the system. We express the cost of the system vs. the required throughput of the system $\lambda$, for constant $T$, in the following way. We substitute $t_S + t_N + t_A = 3t$ and $m = T \lambda$ into Equation 1 and solve by $t$. We obtain $\frac{1}{t} = \frac{3AT}{T - 2d}$ as the cost of each resource.

These results strongly motivate the introduction of a pipelining approach, which is described in Section 7.
6. Overview of Intermediate and Main Results

The blocking protocol is not scalable and provides very weak security guarantees (for example, the server can easily reorder updates and reply on the basis of old versions). In this paper, we provide a scalable and secure protocol that solves the same problem. We incrementally describe the solution in the next three sections. First, we show how it is possible to pipeline requests to the authenticator without waiting for its responses. At this stage, no particular consistency and security are provided. Then, we show a distinct result in which we do not care about efficiency, but we deal with strong consistency and security. Lastly, we show how to combine this two results.

In this section, we informally describe the ideas underlining these results, while complete details and formal proofs are provided in Sections 7, 8 and 9.

6.1. The Simplified Pipeline-Integrity Protocol

Our first objective is to devise a protocol that allows the server to send an authentication request without waiting for the result of the previous one. This is an essential aspect of our pipelining approach. We observe that the only information the authenticator sends back to the server is the signature of the new root-hash. This means that the server may decide to send an authentication request at any time while being able to build it with all the information, as in the blocking approach, except for the signature of the previous root-hash. Hence, our goal is to allow the authenticator to provide a proof that all the checks it performed were successful, without relying on the signature of the previous root-hash. The resulting proof should be usable by the server to build complete authentications to be used, for example, in read replies. In our approach, the authenticator can do that for any kind of checks, no matter how complex they are.

We introduce the concept of conditional authentication, which is formally defined in Section 7. It expresses the fact that a certain root-hash \( r_i \) is correct, if the previous one \( r_{i-1} \) was. Root-hash \( r_i \) results from \( r_{i-1} \) by the application of a sequence of update invocations that passes certain (consistency) checks. A conditional authentication is a signature of the ordered pair of root-hashes \( r_{i-1} \) and \( r_i \). Conditional authentications can be chained with other (conditional or regular) authentications, if certain conditions are met (see Section 7).

In Figure 6, we provide an example of how we use conditional authentications. A regular stream of updates arrives to the server. The server sends...
Figure 6: An example of use of conditional authentications to enable pipelining of authentication requests.
authentication requests to the authenticator at regular intervals of time. In the example, authentication requests are pipelined since, between an authentication request and its reply, the server sends other authentication requests. No root-hash signature is sent in these authentication requests, hence, the authentications contained in the replies are conditioned. In the figure, authentication requests are sent at instants \( t_1, t_2, \ldots \). An authentication request sent at time \( t_i+1 \) includes the updates arrived since the sending of the previous authentication request at time \( t_i \). The state of the dataset right after \( t_i \) is denoted \( D_i \) and its root-hash is denoted \( r_i \). The proofs in the authentication request sent at time \( t_i+1 \) are based on \( D_i \). Suppose that between \( t_5 \) and \( t_6 \) a read invocation is received. To authenticate the root-hash of the proof contained in the response, the server can include the conditioned authentications it received (\( r_2 \) with respect to \( r_1 \) and \( r_1 \) with respect to \( r_0 \)). If the server knows a signature that authenticate \( r_0 \), it can be included with those conditional authentications to provide a chain that has the same semantic of a regular authentication. In Section 7 we formally prove this, we show how to keep the chain bounded, we provide a formal description of our protocol, and we analyse its scalability.

6.2. An ADS-Based Quasi-Fork-Linearisabile Protocol

After having provided a scalable protocol, we focus on consistency and security. We introduce a protocol, called history-integrity protocol, that securely ensures quasi-fork-linearisability (see Definition 5) in the sense that any deviation of the server from that behaviour is detected. We recall that quasi-fork-linearisability is a consistency model in which the server can fork the history of the updates showing distinct branches to distinct clients and where intersection among branches is forbidden, except right after the fork. Essentially, our objective is to fulfil the following security requirements, which are tightly linked with some of the consistency constraints introduced in Section 4.4.

- **R1** Each update should appear exactly once in the sequence of updates to be applied to the dataset. The order chosen by the server should conform to the order each client issued its updates. A violation of this rule by the server must be detected. This requirement is linked with Items 1 and 2 of Definition 5.

- **R2** Clients should be able to detect if the server is trying to propose outdated versions of the dataset. That is, each client \( c \) should check that
each dataset version proposed by the server follows the last one that
$c$ has knowledge of. This requirement is linked with the monotonicity
definition introduced in Section 4.4.

R3 Trusted entities should be able to detect the joining of forks accord-
ing to the definition of quasi-fork-linearisability. That is, overlapping
between distinct forks is allowed only for the updates invoked before
the client receives any response from the server after the fork. This
requirement is linked with Item 4 of Definition 5.

In Section 8, we describe a number of techniques that address the above
requirements in a blocking setting. These techniques turn out to be compat-
ible with the pipelining approach described above. Now, we briefly describe
the intuition underlying those techniques.

Requirement R1 is addressed by hash-chaining the update invocations
of each client and checking the consistency of the chain for each client on
the authenticator. To keep track of the hash of the last update invocation
across consecutive authentications, we authenticate this information in the
very same ADS used to authenticate the dataset, under special client-keys.

Requirement R2 is addressed hash-chaining the root-hashes of consecutive
versions of the dataset. The server responses to read invocations are always
based on a certain version identified by a root-hash. Consider two consecutive
read responses, $\rho_1$ and then $\rho_2$, sent to a client $c$ based on versions identified
by $r_1$ and $r_2$, respectively. In each response, the server provides a proof of
monotonicity. In our example, this is the proof that $r_1$, that $c$ saw in $\rho_1$,
precedes $r_2$ in the hash-chain of the root-hashes. To obtain a short proof, we
adopt an additional history ADS on this hash-chain whose root-hash is itself
authenticated by the authenticator.

To address Requirement R3, each client sends, along with each invocation,
the indication of the last dataset version it knows. The server must include
this information, equipped with a proof obtained from the history ADS, in
any authentication request. This is enough to enable authenticators to detect
violations of the quasi-disjoint-forking rule.

In Section 8, we formally describe the above mentioned techniques and
provide proofs of their security and correctness.

6.3. The Pipeline-Integrity Protocol

In Section 9, we show that it is possible to combine the above results. Even if this is the main result of the paper, the resulting protocol and algo-
The algorithms inherit the technicalities of the previous intermediate results without adding any new fundamental concept. The messaging scheme is the same that we show for the simplified pipeline-integrity protocol, hence, the scalability properties, shown in Section 7, are preserved. Security and correctness of the combined solution, derive from the corresponding security and correctness properties of the history-integrity protocol, proven in Section 8. This extension is possible because of the chaining properties of the conditional authentications, introduced in Section 4.

7. The Simplified Pipeline-Integrity Protocol

From the analysis provided in Section 5, it is evident that the blocking approach obtains very poor results, in terms of throughput or latency of the whole system, compared with the theoretical capability of the distinct elements of the system. We recall that, according to the blocking approach, the rounds of authentication of the root-hashes are executed sequentially and the server blocks until the authentication reply is received (see Section 5).

In this section, we show how it is possible to create a protocol, which we call simplified pipeline-integrity protocol, that achieves much better results by pipelining authentication rounds. For the sake of simplicity, in this section, we focus only on the interaction scheme among the actors, disregarding all security and consistency aspects that are not strictly needed to explain it. Since the guarantees of the simplified pipeline-integrity protocol are quite modest, we do not provide any formal proof about them. A consistent and secure (but inefficient) protocol is shown in Section 8. In Section 9, that protocol is enriched with the interaction scheme shown in this section obtaining consistency, security and efficiency.

In the simplified pipeline-integrity protocol, invocations and responses for read and update operations have format and semantic very similar to those of the blocking approach. The only difference is related to the authentications of root-hashes, which are substituted by chained authentications, introduced in the following section.

7.1. Conditional and Chained Authentications

Consider an authenticator that is performing consistency checks and is computing and signing the new root-hash. At the same time, the server can apply updates it is receiving, creating a new status of the dataset and ADS. We recall that using ADSes, we can efficiently link a cryptographic...
hash, called root-hash, with a large dataset of key-value pairs and that root-hash is supposed to be authenticated (usually signed) by a trusted entity. A fundamental idea of our contribution is that an additional authenticator can conditionally authenticate a root-hash \( r_i \) even if the signature of the previous root-hash \( r_{i-1} \) is not known yet. This is not a real authentication of \( r_i \) but it is still something that can be used together with an authentication of the of \( r_{i-1} \), when it will be available. The simplified pipeline-integrity protocol allows the server to start a new authentication round when the previous one is not finished yet. Actually, the server may create a pipeline of authentication rounds which can be arbitrarily deep. By pipelining authentication rounds, we get three important advantages.

- We make better use of resources, since server, network and authenticators all work in parallel.

- We can achieve a much better trade-off between throughput and latency, since the authentication of a sequence of updates is split into several short rounds that are processed concurrently.

- We can have several authenticators working in parallel, each addressing a different set of updates.

As we will see, the cost to pay for this approach is that additional root-hash signatures have to be enclosed in the messages sent form server to trusted entities. However, this cost turns out to be quite small compared with the large advantages obtained (see Sections 7.5 and 11).

We denote by \( q \) the number of ongoing authentication rounds at steady operational pace. For the sake of simplicity, we assume the \( q \) ongoing authentications are performed with \( q \) different authenticators denoted by \( a_1, \ldots, a_q \), where each authenticator can be in charge of only one authentication request at a time.

In the following, we deal with root-hash authentications in three forms: plain, conditional and chained. The plain authentication was introduced in Section 5 and it consists of just a signature of a root-hash. We call conditional authentication a signed pair \([r_i, r_{i+j}]\) \((j \geq 1)\), whose semantic is the following: \( r_{i+j} \) is authenticated on the basis of data that are supposed to be genuine against \( r_i \), hence, if an authentication for \( r_i \) is provided, also \( r_{i+j} \) can be considered authentic. The first root-hash of the pair is said to be conditioning while the second is said to be conditioned.
Two conditional authentications form a chain if the conditioning root-hash of the second is equal to the conditioned root-hash of the first. The chain of two conditional authentication is written \([r_i, r_j] [r_j, r_l]\) with \(i < j < l\).

**Property 2.** The chain of two conditional authentications \([r_i, r_j] [r_j, r_l]\) is semantically equivalent to the conditional authentication \([r_i, r_l]\).

*Proof.* Consider an authenticator \(a\) generating \([r_i, r_j]\) \(a\). When \(a\) assumes that \(r_i\) is a valid root-hash, it is equivalent to assume that its associated dataset \(D_i\) complies to a number of consistency rules. We can summarise this saying that a certain logic predicate \(p_i\) about \(D_i\) is true. Equivalently, stating that \(r_j\) is a valid, is equivalent to stating that predicate \(p_j\) is true. Hence, when \(a\) signs the pair \((r_i, r_j)\), it states that the logic formula \(p_i \rightarrow p_j\) holds. Analogously, \([r_j, r_l]\) and \([r_i, r_l]\) are equivalent to stating that \(p_j \rightarrow p_l\) and \(p_i \rightarrow p_l\) holds, respectively. By the rules of predicate logic, \((p_i \rightarrow p_j) \land (p_j \rightarrow p_l)\) entails \(p_i \rightarrow p_l\). □

**Property 3.** A plain authentication \([r_i]\) with the conditional authentication \([r_i, r_j]\) is semantically equivalent to the plain authentication \([r_j]\).

*Proof.* Consider an authenticator \(a\) generating \([r_j]\) \(([r_i])\). Before signing it, \(a\) checks that \([r_j]\) \(([r_i])\) complies to a number of consistency rules, which can by summarised by logic predicate \(p_i\) \((p_j)\) about dataset \(D_i\) \((D_j)\). Also, signing \([r_i, r_j]\), is equivalent to state \(p_i \rightarrow p_j\) (see proof of Property 2). By the rules of predicate logic, \(p_i \land (p_i \rightarrow p_j)\) entails \(p_j\). □

Properties 2 and 3 justify the extension of the definition of chained authentication to a sequence starting with one plain authentication followed by several chained conditional authentications. For example, from the above definitions, the sequence \([r_0] [r_0, r_5] [r_5, r_9]\) is a chained authentication, which, by Properties 2 and 3, is semantically equivalent to \([r_9]\).

Authentication chains can be arbitrarily long. Before trusting a chained authentication, we should check its coherency according to the above properties and verify its signatures. This procedure is formalised by Algorithm 1. We call compaction the process of reducing an authentication chain into a plain authentication, like \([r_9]\). This process is formalised by Algorithm 2.

Compaction should be performed by a trusted entity, since the final result requires a signature. In the simplified pipeline-integrity protocol, compaction is performed by authenticators.
Algorithm 1 Verification of a chained authentication.

**Input:** a chained authentication $A = [r_0] [r_0, r_1] \ldots [r_{n-1}, r_n]$ and the sequence of root-hashes $\bar{A} = r_0, r_1, \ldots, r_n$. In the algorithm, we refer to elements of $A$ as the 0-th, 1-st, $\ldots$, $n$-th.

1: Check the signature $[r_0]$ of $r_0$
2: for $i$ in $1 \ldots n$ do
3: Check the signature $[r_{i-1}, r_i]$ of the pair $(r_{i-1}, r_i)$
4: if $i = 1$ then
5: Let the initial elements of $A$ be $[y][x, r_1]$
6: Check that $x = y$
7: else
8: Let the $(i-1)$-th and the $i$-th elements of $A$ be $[r_{i-2}, y][x, r_i]$
9: Check that $x = y$
10: end if
11: end for
12: $A$ and $\bar{A}$ are verified if all the above checks are successful.

Algorithm 2 Compaction of a chained authentication.

**Input:** a chained authentication $A = [r_0] [r_0, r_1] \ldots [r_{n-1}, r_n]$ and the sequence of root-hashes $\bar{A} = r_0, r_1, \ldots, r_n$.

1: Perform Algorithm 1 on $A$ and $\bar{A}$.
2: if $A$ and $\bar{A}$ are successfully verified then
3: return $[r_n]$
4: else
5: fail
6: end if
7.2. Pipelined Execution of Authentication Requests

The adoption of chained authentication allows us to pipeline authentication requests. In Section 5, a commit starts when an authentication request is sent and ends at the reception of the corresponding reply. While waiting for the reply, the server cannot do anything. According to definitions in Section 4.3, the server cannot even reply to read operations. However, nothing prevents to think about commits as if they were limited to the processing of the authentication reply only. In this way, the server can, at least, reply to read requests on the basis of the previous state of the dataset, while waiting for the authentication reply. We now go further showing how it is possible to start an authentication request before receiving the response to the previous one.

To simplify the explanation, in the following, we assume all processing time on server and authenticators to be negligible as well as transmission time, but we assume the one-way transmission delay to be non negligible and denoted by $d$. These hypotheses are relaxed at the end of this section when we evaluate the scalability of the protocol. A commit $\chi_i$ encompasses all the operations needed (on server or authenticator) to authenticate version $i$ of the dataset, which we denote $D_i$. We denote $D_0$ the initial state of the dataset. An ADS on $D_i$ is denoted by $\Delta_i$ and its root-hash $r_i$. The reader should consider all these symbols as abstract mathematical values. The state of the server will be introduced later. We call $t_i$ the instant when the commit request $\rho_i$ related to $\chi_i$ is sent, which contains all information needed by the authenticator to compute authentication for $D_i$. We associate with $\chi_i$ all updates whose requests are received in the interval $(t_{i-1}, t_i)$.
### Figure 7: An example of execution of the simplified pipeline-integrity protocol with three authenticators.

<table>
<thead>
<tr>
<th>Time</th>
<th>Server Side</th>
<th>Authenticators Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0$</td>
<td>$D_0$</td>
<td>$a_0$</td>
</tr>
<tr>
<td>$t_1$</td>
<td>$D_1$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$D_2$</td>
<td>$a_2$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$D_3$</td>
<td>$a_3$</td>
</tr>
<tr>
<td>$t_4$</td>
<td>$D_4$</td>
<td>$a_4$</td>
</tr>
<tr>
<td>$t_5$</td>
<td>$D_5$</td>
<td>$a_5$</td>
</tr>
<tr>
<td>$t_6$</td>
<td>$D_6$</td>
<td>$a_6$</td>
</tr>
<tr>
<td>$t_7$</td>
<td>$D_7$</td>
<td>$a_7$</td>
</tr>
<tr>
<td>$t_8$</td>
<td>$D_8$</td>
<td>$a_8$</td>
</tr>
<tr>
<td>$t_9$</td>
<td>$D_9$</td>
<td>$a_9$</td>
</tr>
<tr>
<td>$t_{10}$</td>
<td>$D_{10}$</td>
<td>$a_{10}$</td>
</tr>
</tbody>
</table>

$\rho$ contains Chain:

- $[r_0, r_1, \ldots, r_q]$  
- $[r_1, r_2, \ldots, r_q]$  
- $[r_2, r_3, \ldots, r_q]$  
- $[r_3, r_4, \ldots, r_q]$  
- $[r_4, r_5, \ldots, r_q]$  
- $[r_5, r_6, \ldots, r_q]$  
- $[r_6, r_7, \ldots, r_q]$  
- $[r_7, r_8, \ldots, r_q]$  
- $[r_8, r_9, \ldots, r_q]$  
- $[r_9, r_{10}, \ldots, r_q]$  

$\rho_{pl}$ contains Compacted:

- $[r_0, r_1]$  
- $[r_1, r_2]$  
- $[r_2, r_3]$  
- $[r_3, r_4]$  
- $[r_4, r_5]$  
- $[r_5, r_6]$  
- $[r_6, r_7]$  
- $[r_7, r_8]$  
- $[r_8, r_9]$  
- $[r_9, r_{10}]$

### Figure 8: The general scheme that links authentication requests/replies to the changes of the state of the server in the simplified pipeline-integrity protocol.

We assume synchronous and reliable operation (these hypotheses are relaxed in Section 10). Figure 7 depicts the communication between the server and $q$ authenticators (where $q = 3$ in the figure) for the simplified pipeline-integrity protocol. To have a pipeline with $q$ stages (one for each authenticator), the server must send an authentication request every $\Delta t = t_i - t_{i-1} = 2d/q$ for all $i > 2$. Let $\rho_{pl}^i$ be the reply to authentication request $\rho_i$. From the above assumptions, starting from $t_{q+1}$, $\rho_{pl}^i$ is received.
at \( t_{i+q} \). As detailed below, \( \rho_i^{\text{rpl}} \) contains conditional authentication \([r_{i-1}, r_i]\) to be used to conditionally authenticate \( D_i \). This is computed on the basis of values taken from \( D_{i-1} \), of proofs derived from \( \Delta_{i-1} \), and of all updates arrived at the server between \((t_{i-1}, t_i)\). We assume that \( D_0 \) is empty and the corresponding \( r_0 \) is authenticated by \( A = [r_0] \) know by the server. When \( \rho_i^{\text{rpl}} \) is received, the conditional authentication \([r_{i-1}, r_i]\) is appended by the server to \( A \) to obtain the chained authentication of \( D_i \). To avoid that \( A \) grows indefinitely, the server includes into \( \rho_i^{\text{rpl}} \) the current content of \( A \). The authenticator performs its compaction, obtaining \([r_{i-q}]\), and includes it into \( \rho_i^{\text{rpl}} \). The server uses \([r_{i-q}]\) to shorten \( A \). This is done starting from \( t_{2q+1} \). In this way, \( A \) turns out to be bounded in length.

An authentication request is outstanding if no corresponding reply was received for them yet. In our setting, at most \( q \) authentication requests can be outstanding.

7.3. Server Data Structures

To realize the simplified pipeline-integrity protocol, the server keeps two notable categories of data structures. They are \( R\)-data-structures and \( U\)-data-structures. They are distinguished by superscript \( R \) and \( U \) respectively. The first category is dedicated to serving read invocations, the second is dedicated to the processing of update invocations. For the simple pipeline-integrity protocol, the following are parts of the status of the server: dataset \( D^R \) with its ADS \( \Delta^R \) and dataset \( D^U \) with its ADS \( \Delta^U \), which are stored by the server.

These data structures change value only at instants \( t_i \). The values assumed by each data structure between instants \( t_i \) and \( t_{i+1} \) is denoted by the same symbol, with subscript \( i \), like \( D^R_i \) and \( \Delta^R_i \). We denote by \( l \) the index of the last time instant \( t_l \) in which an update request was sent, which is also the last instant in which the status was updated. In the absence of subscript, current value is assumed, for example, \( D^R = D^R_l \) and \( D^U = D^U_l \).

For the hypothesis of synchronous operation with \( \Delta t = 2d/q \), \( \rho_i^{\text{rpl}} \) is always received at \( t_{i+q} \). At \( t_l \) (see Figure 8), an authentication reply \( \rho_{l-q}^{\text{rpl}} \) is received and R-data-structures are updated to version \( D^R_l \) and \( \Delta^R_l \) on the basis of the updates contained in \( \rho_{l-q} \), i.e., contained in the corresponding request. Further authentication request \( \rho_l \) is sent containing proofs based on \( D^U_{l-1} \) and \( \Delta^U_{l-1} \). Each R-data-structure tracks the corresponding U-data-structure with a delay of \( q\Delta t \) and the following hold: \( D^U = D_l \), the root-hash of \( \Delta^U_i \) is \( r_i \), \( D^R = D^U_{l-q} \), \( \Delta^R = \Delta^U_{l-q} \), the root-hash of \( \Delta^R_i \) is \( r_{i-q} \).
Table 1: State of the server for the simplified pipeline-integrity protocol.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D^R$</td>
<td>Dataset used to serve read invocations.</td>
</tr>
<tr>
<td>$\Delta^R$</td>
<td>ADS related to $D^R$.</td>
</tr>
<tr>
<td>$D^U$</td>
<td>Dataset used to record updates and send authentication request.</td>
</tr>
<tr>
<td>$\Delta^U$</td>
<td>ADS related to $D^U$.</td>
</tr>
<tr>
<td>$l$</td>
<td>Index of the dataset version contained in $D^U = D_l$.</td>
</tr>
<tr>
<td>$A$</td>
<td>Authentication of $D^R$ and $\Delta^R$ in the form $[r_{l-2q}, r_{l-2q+1}]$</td>
</tr>
<tr>
<td></td>
<td>$[r_{l-q-2}, r_{l-q-1}]$ $\ldots$ $[r_{l-q-1}, r_{l-q}]$.</td>
</tr>
<tr>
<td>$\bar{A}$</td>
<td>Sequence of the root-hashes $r_{l-2q}, r_{l-2q+1}, \ldots, r_{l-q}$ on which $A$ is based.</td>
</tr>
<tr>
<td>$\Omega$</td>
<td>Queue of outstanding authentication requests $\rho_{l-q}, \ldots, \rho_l$</td>
</tr>
<tr>
<td>$Q^c$</td>
<td>A queue for each client $c$ containing update invocations of $c$ for which no authentication request was sent yet.</td>
</tr>
</tbody>
</table>

Even if theoretically we say that the server keeps $D^R (\Delta^R)$ and $D^U (\Delta^U)$, since the first is a delayed version of the second, they only differ for the updates arrived after $t_{l-q}$. Efficient storage solutions can be devised to do that without doubling space occupation.

Additionally, the server keeps

- a chained authentication $A$ for $D_l^R$ with the following structure $[r_{l-2q}]$ $[r_{l-2q}, r_{l-2q+1}]$ $[r_{l-q-2}, r_{l-q-1}]$ $\ldots$ $[r_{l-q-1}, r_{l-q}]$ and the corresponding sequence of root-hashes $\bar{A} = r_{l-2q}, r_{l-2q+1}, \ldots, r_{l-q}$,

- a queue $\Omega$ of all outstanding authentication requests, which, after $t_{q+1}$, is $\rho_{l-q}, \ldots, \rho_l$, and

- for each client $c$, a queue $Q^c$ containing all the update operations invoked by client $c$ and received by the server (in the invocation order) that are not associated with a commit, i.e. that have not been sent in an authentication request, yet.

Table 1 summarises the content of the state of the server for the simplified pipeline-integrity protocol.

### 7.4. Authentication: Messages and Processing

In the simplified pipeline-integrity protocol, authenticators do not keep any state. The server sends an authentication request $\rho_l$ at time $t_l$ with the
1. getting what is missed in $A$ to get a chained authentication of $D_l$ (containing updates invocation arrived up to $t_l$), which will be the content of $D^R$ after the reception of $\rho_{l}^{\text{pl}}$, and

2. getting a compacted version of the current $A$, which will be equal to $[r_{l-2q}]$ after the reception of $\rho_{l}^{\text{pl}}$.

Authentication request $\rho_{l}$ sent at time $t_l$ contains

- a sequence of all update operations received by the server between $t_{l-1}$ and $t_l$ (currently stored in the queues $Q^c$), preserving the order that they have in $Q^c$, where the interlacing of the sequences of updates of distinct clients is arbitrarily chosen by the server,

- the proofs for all $\langle k, v \rangle$ involved in the above updates, computed according to $\Delta_{l-1}$, where $v$ is the value as in $D_{l-1}$,

- the current authentication chain $A$, with the corresponding root-hashes $A$, to be compacted.

Upon reception of an authentication request, the authenticator $a$ performs the actions described in Algorithm 3. Since the server does not provide authentication for $\bar{r}$, the authenticator only provides a conditional authentication of the subsequent root-hash $\bar{r}$ on the basis of the assumption of authenticity of $\bar{r}$. Proving the authenticity of $\bar{r}$ is up to the trusted entity that will use that conditional authentication. In our approach, this is essentially done considering the conditional authentication within the context of an authentication chain.

Supposing synchronous operation, $\rho^{\text{pl}}$ reaches the server after $q\Delta t$ with respect to the instant $\rho$ was sent. This means that at each $t_l$ the server gets $[r_{l-2q}]$ and $[r_{l-q-1}, r_{l-q}]$. These values are used by the server to update $A$ so that current value of $D^R$ can be authenticated with a chain of $q$ conditional authentications plus one plain authentication.

When the server receives $\rho^{\text{pl}}$, it executes Algorithm 4 to update its state and to send the new authentication request. The algorithm starts its execution at $t_l$ by incrementing the variable $l$. Lines 2-4 are related to the processing of $\rho_{l-1-q}^{\text{pl}}$. After them, the following read invocations are served on the basis of $D^R = D_{l-q}$. Lines 5-11 are related to the creation of $\rho_{l}$ on the basis of $D^U = D_{l-1}$ and to the update of $D^U$ to be ready for $t_{l+1}$.

Purpose of

R2.15
subsection
removed
Algorithm 3  Simplified pipeline-integrity protocol – Authenticator. Operations performed by an authenticator $a$ upon reception of an authentication request $\rho$.

**Input:** An authentication request $\rho$ that was sent by the server at time $t_l$, containing:

- a sequence $B$ of updates in the form $u^c = \langle k, v' \rangle$, where $c$ is the client that invoked the update and $v'$ is the new value of $k$,
- for all keys $k$ involved in $B$, proof($\langle k, v \rangle$) where $v$ is the previous value of $k$,
- chained authentication $A = [r_{i-q}, r_{i-q}, r_{i-q+1}] \ldots [r_{i-1}, r_{i}]$ and corresponding sequences of root-hashes $\bar{A}$ (see the status of the server in Table 1).

1: Arbitrarily select one of the proofs and compute the root-hash $\bar{r}$.
   $\triangleright \bar{r}$ is supposed to match $r_{l-1}$ on the server when $\rho$ is sent.

2: Check all other proofs against $\bar{r}$ to verify that they all come from the same dataset version.

3: Computes from the proofs and from new values, the new root-hash $\tilde{r}$.
   $\triangleright \tilde{r}$ is supposed to match $r_{l-q}$ on the server when $\rho_{\text{rpl}}$ is received.

4: Sign the conditional authentication $[\bar{r}, \tilde{r}]_a$.

5: Based on $A$ and $\bar{A}$, compute a compact version $[r_i]_a$ of $A$.
   $\triangleright i$ turns out to be $l - 2q$ when $\rho_{\text{rpl}}$ is received.

6: Sends the authentication reply $\rho_{\text{rpl}}$ containing $[\bar{r}, \tilde{r}]_a$ and $[r_i]_a$. 

38
Algorithm 4 Simplified pipeline-integrity protocol – Server. Operations performed by the server upon reception of an authentication reply.

**Input:** An authentication reply $\rho^{\text{pl}}$ from authenticator $a$ containing $[r_{l-q-1}, r_{l-q}]$ and $[r_{l-2q}]$.

1: $l \leftarrow l + 1$
2: Pull from $\Omega$ the authentication request $\rho$ corresponding to $\rho^{\text{pl}}$.
   $\triangleright$ $\rho$ should be the first in $\Omega$, due the timing hypothesis.
3: Update $D^R$ and $\Delta^R$ according to the update operations specified in $\rho$.
4: Update $A$ using authentications of $\rho^{\text{pl}}$, namely, $[r_{l-q-1}, r_{l-q}]$ is added to the right of $A$ and $[r_{l-2q-1}, r_{l-2q}]$ is substituted by $[r_{l-2q}]$.
   $\triangleright$ $A$ should turns out to be the authentication of current $D^R$.
5: Let $Y$ be an empty sequence of updates.
6: **for each** client $c$ **do**
7: Pull from $Q^c$ all updates and append them to $Y$ in the same order.
   $\triangleright$ The interlacing of updates of different clients may be arbitrarily chosen
8: **end for**
9: Prepare a new authentication request $\rho'$ for $a$, containing all updates in $Y$ with their signatures, proofs computed according to the current value of $D^U$ and $\Delta^U$, new values, and the current value of $A$ to be compacted.
10: Push $\rho'$ as last element of $\Omega$.
11: Send $\rho'$ to $a$.
12: Update $D^U$ and $\Delta^U$ according to the updates of $Y$. 
client(s)

server

time

authenticator(s)

\[ q = T \]
\[ \Delta t = 6 \]

Each authentication round handles \( m = \lambda \Delta t \) updates.

Figure 9: Interaction and timings between client(s), server, and authenticator(s) for the simplified pipeline-integrity protocol, when non-negligible computation and transmission time are considered.

7.5. Scalability

With the intent to evaluate the scalability of the pipeline-integrity protocol, we relax the hypothesis of negligible computation and transmission time, but we keep operations synchronous. Essentially, we put ourselves in a setting comparable with the setting shown in Section 5. As stated in Section 4.2, ideally we would like to achieve high throughput while keeping response time bounded. More formally, let \( \lambda \) be the arrival rate of the updates and \( d \) be the one-way network delay between server and authenticators. Our ideal scalability objective is to have response time \( O(d) \), that is, independent from how \( \lambda \) is large. We now show that with the above described protocol we can get very close to the ideal goal.

Let \( t_S \), \( t_N \), and \( t_A \) be the time taken to process or transmit one update operation during one authentication round by the server, the network and the authenticator, respectively. Let \( \hat{t}_S \), \( \hat{t}_N \), and \( \hat{t}_A \) the time taken for processing or transmitting one conditional authentication of the authentication chain by the server, the network and the authenticator, respectively. Let \( \tilde{t}_S \), \( \tilde{t}_N \), and \( \tilde{t}_A \) a constant amount of time spent in a round by the server, the network and the authenticator, respectively.

Figure 9 depicts this new setting, where \( T \) is the time taken by each
authentication round, $\lambda$ is the arrival frequency of the updates, $d$ is the one-way network delay between server and authenticators, $q$ is the number of authenticators (which equals the length of the authentication chain), $\Delta t = T/q$ is the interval of time between the start of two consecutive authentication rounds, and $m = \lambda \Delta t$ is the number of updates to be processed by one round.

We denote by $\alpha = t_S + t_N + t_A$ the total processing/transmission time for one update, by $\beta = \tilde{t}_S + \tilde{t}_N + \tilde{t}_A$ the total processing/transmission time for one conditional authentication, and by $\gamma = \bar{t}_S + \bar{t}_N + \bar{t}_A + 2d$ the constant terms.

Now, suppose to keep $\alpha$, $\beta$, $\gamma$ and $\lambda$ constant and to increase $q$, while correspondingly decreasing $\Delta t$ and $m$, and to observe how the duration of a round varies. The duration of a round is

$$T = m\alpha + q\beta + \gamma. \quad (3)$$

Substituting $m = \lambda T/q$ and solving by $T$, we have

$$T = \frac{q\beta + \gamma}{1 - \frac{\alpha\lambda}{q}}. \quad (4)$$

We consider $T$ as a function of $q$, defined for $q \in (\alpha\lambda, +\infty)$, and find the value of $q$ for which $T(q)$ is minimum. By regular calculus, the minimum is reached at

$$q_{\min} = \alpha\lambda + \sqrt{(\alpha\lambda)^2 + \frac{\gamma\alpha\lambda}{\beta}}. \quad (5)$$

For $q \in (\alpha\lambda, q_{\min})$, $T(q)$ is decreasing. For $q > q_{\min}$, $T(q)$ is above the asymptote $T = \beta q + \gamma$. By simple substitution, it is easy to see that $T(q_{\min})$ is not bounded by a constant when $\lambda$ increases. However, by monotonicity of square root, $2\alpha\lambda < q_{\min}$ and $T(q_{\min}) < T(2\alpha\lambda) = 4\beta\alpha\lambda + 2\gamma$, which is a line with a very small slope, since $\alpha$ and $\beta$ are usually quite small compared to $\gamma$ (see below). One may object that $q_{\min}$ is fractional, in general, and we are forced to choose either $\lfloor q_{\min} \rfloor$ or $\lceil q_{\min} \rceil$. It is easy to show that $q_{\min} - 2\alpha\lambda > 1$ when $\lambda > \frac{1}{\alpha(\gamma/\beta - 2)}$. Hence, for $\lambda$ large enough, we have $2\alpha\lambda \leq \lfloor q_{\min} \rfloor \leq q_{\min}$. In this case, since $T(q)$ is decreasing up to $q_{\min}$, we obtain $T(\lfloor q_{\min} \rfloor) \leq T(2\alpha\lambda) = 4\beta\alpha\lambda + 2\gamma$.

The above arguments support the following theorem.
Theorem 1 (Scalability). In the simplified pipeline-integrity protocol, there exists a value of the number of authenticators \(q\) for which the response time for each authentication request is bounded by \(4\beta \alpha \lambda + 2\gamma\) for \(\lambda\) large enough, where \(\alpha\) is the total processing/transmission time for one update, \(\beta\) is the total processing/transmission time for one conditional authentication, and \(\gamma\) is the remaining processing/transmission time and network delay in a round that does not depend on the number of updates in the request or \(q\).

We point out that the results stated by Theorem 1 is very close to our ideal objective. In fact, from measurements performed contextually to the experiments of Section 11, we observed that the product \(\alpha \beta\) is in the order of \(10^{-6}\) and \(\gamma\) is in the order of \(10^{-1}\) seconds. For the two terms to be comparable \(\lambda\) should be in the order of \(10^5\) updates per seconds. If we consider negligible the first term, \(T\) is \(O(\gamma)\), where \(\gamma\) is largely dominated by \(d\).

The scalability of the simple-pipeline integrity protocol is further supported by the following analyses.

The presence of the authentication chain introduce some overhead. Every authentication request carries an authentication chain of length \(q\). We have \(q\) rounds every \(T\), hence, the overhead introduced on the work of authenticator, network, and server is \(q^2 \tilde{t}_A / T\), \(q^2 \tilde{t}_N / T\), and \(q^2 \tilde{t}_S / T\), respectively. In practice, we expect \(q\) to be small (it is less than 8 in the experiments of Section 11). Further, \(\tilde{t}_A\), \(\tilde{t}_N\), and \(\tilde{t}_S\), depend on the choice of cryptographic primitives, but we do not expect them to be much larger than \(t_A\), \(t_N\), and \(t_S\), respectively. Hence, we expect the overhead to be quite small in practice.

The bottleneck of the system is either the authenticator or the network or the server. Supposing \(\beta q\), \(\tilde{t}_S\), \(\tilde{t}_N\), and \(\tilde{t}_A\) to be small, the throughput of the system is approximatively given by \(\min \left( \frac{1}{\tilde{t}_S}, \frac{1}{\tilde{t}_A}, \frac{1}{\tilde{t}_N} \right)\). Supposing the resources to be perfectly balanced, we can state \(t = t_A = t_N = t_S\). Under this assumption, when the system is computing at its maximum speed, all resources are fully busy. This is much better than what we noted for the blocking approach (Section 5), which heavily underutilises resources.

Now, we aim to understand how much this solution costs in terms of additional throughput to be provisioned to resources in order to increase the maximum throughput of the system while keeping \(T\) constant. We additionally assume the time spent to perform distinct activities on the same resource to be proportional with the same factor. This allows us to state \(\alpha = 3t\), \(\beta = 3at\), and \(\gamma = 3bt\), where \(a\) and \(b\) are constants. In this way, \(1/t\) is proportional to both the throughput of each resource and to the cost of
<table>
<thead>
<tr>
<th></th>
<th>Blocking</th>
<th>Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maximum throughput</strong></td>
<td>$\frac{1}{t_S + t_N + t_A}$ theoretical since $T \to \infty$</td>
<td>$\min \left( \frac{1}{t_S}, \frac{1}{t_N}, \frac{1}{t_A} \right)$ approximated</td>
</tr>
<tr>
<td><strong>Round duration</strong> (proportional to response time)</td>
<td>$\frac{2d}{1 - \alpha \lambda}$ $\alpha = t_S + t_N + t_A$</td>
<td>$\frac{q\beta + \gamma}{1 - \frac{\alpha \lambda}{q}}$ $\alpha = t_S + t_N + t_A$ see text for definitions of $\beta$, $\gamma$, and $q$</td>
</tr>
<tr>
<td><strong>Unused fraction of each resource</strong></td>
<td>$1 - \lambda t$ where $t = t_S = t_N = t_A$ for $t \to 0$, $1 - \lambda t \to 1$ and $T \to 2d$</td>
<td>0 for $t_S = t_N = t_A$ at maximum throughput</td>
</tr>
<tr>
<td><strong>Overhead</strong></td>
<td>0</td>
<td>$\frac{q^2 T_S}{T} \cdot \frac{q^2 T_N}{T} \cdot \frac{q^2 T_A}{T}$</td>
</tr>
<tr>
<td><strong>Cost vs. throughput</strong></td>
<td>$\frac{1}{\frac{1}{t} = \frac{3\lambda T}{T - 2d}}$ for $t = t_S = t_N = t_A$</td>
<td>$\frac{1}{t} = \frac{3\lambda T}{q + aq + b}$ for $t = t_S = t_N = t_A$, see text for the definition of $a$ and $b$</td>
</tr>
</tbody>
</table>

Table 2: Summary of scalability analysis results for the pipelining approach compared against the same results for the blocking approach.
the whole system. From Equation 3 we obtain

$$\frac{1}{t} = \frac{3\lambda T/q + aq + b}{T - 2d}.$$ 

Essentially, supposing $aq + b$ to be small, the simplified pipeline-integrity protocol cuts by $q$ the cost to increase the throughput of the system by a given amount, with respect to the same cost for the blocking approach.

Table 2 summarises the above results and compares them with those obtained in Section 5 for the blocking approach.

8. An ADS-Based Quasi-Fork-Linearisable Protocol

In this section, we show a protocol named history-integrity protocol that provides quasi-fork-linearisability and allows clients to detect deviation from it. In more formal terms, it emulates a key-value store with quasi-fork-linearizability on a Byzantine server. We recall that quasi-fork-linearisability is a consistency model in which the server can fork the history of the updates showing distinct branches to distinct clients. Intersection among branches is ruled out, except for a limited number of updates right after the fork (see Section 4.4).

This protocol does not have the scalability of the simplified pipeline-integrity protocol shown in Section 7, but its construction turns out to be compatible with that approach and it is a fundamental part of the main result of this paper shown in Section 9.

In the rest of this section, we refer to the requirements introduced in Section 6.2. We recall that Requirement R1 is about ensuring that the server does not reorder the updates of each client, Requirement R2 is about ensuring that the server cannot go back in time with the version of the dataset, and R3 is about ensuring quasi-fork-linearisability.

8.1. Server Status

The status of the server for the history-integrity protocol is fully summarised in Table 3. It contains a key-value store $D$ and an ADS $\Delta$ over $D$. Values of $D$ and $\Delta$ change at each commit. We denote by $D_j$ and $\Delta_j$ their value after the $j$-th commit, where $j$ is the version index of the dataset. The root-hash of $\Delta_j$ is denoted $r_j$. The index $l$ of the version that was produced in the last commit is also part of the state of the server. In principle one may expect the authentication $[r_l]$ of the current version of $D$ (and $\Delta$) to be
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>The dataset</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>ADS related to $D$.</td>
</tr>
<tr>
<td>$l$</td>
<td>Index of the version stored in $D$.</td>
</tr>
<tr>
<td>$Q^c$</td>
<td>A queue for each client $c$ containing update invocations of $c$ still not associated with a commit.</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>History ADS, for the authentication of the sequence of history-pairs $\langle j, H_j \rangle$ (see text).</td>
</tr>
<tr>
<td>$P$</td>
<td>A mapping from each client $c$ to a queue of history-pairs, ordered from head to tail by increasing version index.</td>
</tr>
<tr>
<td>$A$</td>
<td>Authentication of $D$, $\Delta$ and $\Pi$ in the form $[R_l]$ where $R_l$ is the root-hash of the current value of $\Pi$.</td>
</tr>
</tbody>
</table>

Table 3: State of the server for the history-integrity protocol.

also stored by the server. However, as will be clear in the following, this is not necessary. The other elements of the server state are introduced in the rest of the description.

Following the client-server interaction described in Section 4.3, the server replies to read requests immediately, if no commit is ongoing. As in the blocking approach (see Section 5), it accumulates the update requests received by client $c$ in a queue $Q^c$. The authentication request for the $j$-th commit contains a sequence of updates in the order they are supposed to be applied to $D_{j-1}$ to obtain $D_j$. The next root-hash $r_j$ is computed by an authenticator on the basis of this sequence and of the proofs of the modified keys, which are preventively checked against $[r_{j-1}]$.

8.2. Consistency Enforcement

To fulfil Requirement R1, we introduce the following construction. Each update operation invoked by client $c$ is represented as a tuple $u(i) = \langle k_i, v_i, \text{hash}(u(i - 1)), i \rangle$ (this is enriched below to satisfy further requirements). When useful, we specify also the client as superscript writing $u^c(i)$. We assume each client specifies a sequence number $i$ for each update, independently from other clients, starting from $i = 0$. In the update invocation, the client sends $u(i)$ along with its signature $[u(i)]_c$. Each client keeps $\text{hash}(u(i))$ to be used in the construction of $u(i+1)$. The only exception is the first update invocation which is $u(0) = \langle k_0, v_0, \eta_0^c, 0 \rangle$, where $\eta_0$ is a constant that is different for each
client (e.g., a random number locally generated by c) and play the role of hash(u(−1)). Clearly, it is possible to check the integrity of a sub-sequence of update invocations u(i), u(i + 1), . . . provided that hash(u(i − 1)) is known. Suppose that an authentication round commits, for a certain client c, updates up to u^c(i). At the next authentication round, we call past-hash for c the value η^c = hash(u^c(i)), that is the hash of the last update that was committed. To check the correctness of the sequence of the updates specified in an authentication request for each client c, an authenticator needs η^c and a way to authenticate it. We introduce special client-keys, one for each client, denoted κ^c. We store the pairs ⟨κ^c, η^c⟩, for each c, in D so that they can be authenticated, as if they were regular data. The initial state of the dataset D = D_0 stored by the server does not contain any regular key but contains all ⟨κ^c, η_0^c⟩ for each client c. Pairs ⟨κ^c, η^c⟩ are sent with proof(Δ, ⟨κ^c, η^c⟩) and are used by authenticators to verify the sequence of u^c(i) specified in the authentication request. During each commit, authenticators also consider the update of η^c when computing r_j. The effective update of ⟨κ^c, η^c⟩ in D_j (and in Δ_j) is performed by the server, as for regular keys.

To fulfil Requirement R2, we introduce the concept of history-hash. After the j-th commit, the history-hash H_j is defined as H_j = hash(H_{j-1}|r_j) (we assume H_{−1} to be an arbitrary constant value to initialise the chain). Clearly, H_j uniquely identifies a sequence of root-hashes and hence a sequence of datasets, up to D_j. We also consider pairs ⟨j, H_j⟩, that we call history-pairs. These are stored in an ADS on the server that we call Π, ordered according to increasing j. See Figure 10 for a picture representing this construction.
The state of this ADS also changes at each commit, hence the state of $\Pi$ after the $j$-th commit is denoted $\Pi_j$. Its root-hash is called history root-hash and denoted $R_j$. The current history root-hash, between two commits, is $R_l$. Its authentication $[R_l]$ is stored by the server in $A$. Note that, to authenticate a key-value pair in the current $D = D_l$ by a proof obtained from $\Delta = \Delta_l$, we do not need to store $[r_l]$. In fact, $[R_l]$ is enough: to authenticate a key-value pair $p$ in $D_l$, we need $p, l, \text{proof}(\Delta_l, p)$, $H_l$-1, $\text{proof}(\Pi_l, \langle l, H_l \rangle)$, and $A = [R_l]$. Each time the server sends a response to a client, it includes this information. The verification procedure is a naive variation of the procedure described in Section 3.

Each client $c$ stores a queue $\Gamma$ of history-pairs, ordered from head to tail in increasing value of version index. We call them local history-pairs for $c$ and are history-pairs that $c$ received from the server. In other words, if $c$ receives a response based on $p = \langle l, H_l \rangle$, where $l$ is the last committed version, $c$ should push $p$ into $\Gamma$, at some point. The server always equips each response with an additional history-pair $\langle V_c, H_c \rangle$ which should be a local history-pair of $c$. The server also includes $\text{proof}(\Pi_l, \langle V_c, H_c \rangle)$ in the response messages. When receiving a message from the server, $c$ always checks that $\langle V_c, H_c \rangle \in \Gamma$, both $p$ and $\langle V_c, H_c \rangle$ are authenticated by $[R_l]$, and $V_c \leq l$. Finally, $c$ pushes $\langle l, H_l \rangle$ into $\Gamma$. Further details are given in Section 8.3.

The server sends to authenticator $a$ an authentication request containing $l - 1$, $H_{l-2}$, $\text{proof}(\Pi_{l-1}, \langle l - 1, H_{l-1} \rangle)$, and $A = [R_{l-1}]$. Authenticator $a$ computes $R_l$ and sends $[R_l]_a$ to the server. In computing $R_l$, besides regular key-value updates and updates of past-hashes for client-keys, $a$ consider also $\Pi_l$ deriving from $\Pi_{l-1}$ by adding $\langle l, H_l \rangle$, where $H_l = \text{hash}(H_{l-1} | r_l)$, as by Figure 10. To enable monotonicity checks by authenticators and to fulfil Requirement $\text{R3}$, we slightly modify the format of updates as follows: $u^c(i) = \langle k_i, v_i, \text{hash}(u(i - 1)), i, \langle V_c, H_c \rangle \rangle$. With respect to the definition of $u^c(i)$ given above, we add the tuple $\langle V_c, H_c \rangle$, that is, the latest history-pair pushed into $\Gamma$ by $c$. Each $u^c(i)$ is put into an authentication request by the server along with the corresponding proof $\langle \Pi, \langle V_c, H_c \rangle \rangle$. With this information, $a$ can perform an additional check to verify that $\langle V_c, H_c \rangle$ is authentic with respect to the history root-hash that $A$ is authenticating and $V_c \leq l - 1$. This is enough to detect violations of the quasi-disjoint-forking rule of Definition 5 (see the proof of Theorem 2) and of monotonicity. In Section 8.3, we provide further details about management of $\Gamma$ and $\Pi$ so that storage is kept bounded and both server and clients always store the needed information to perform the above operations.
We now formally prove some fundamental properties of the history-integrity protocol.

**Property 4** (Monotonicity). *In an execution of the history-integrity protocol in which no trusted entity detects any tampering, the sequence of update operations seen by each client \( c \) monotonically grows.*

*Proof.* Note that, by construction and by security of the cryptographic hash, \( R_l \) is uniquely associated with a sequence of history-pairs and, in turn, to a sequence of updates. When a client \( c \) receives, from the server, a response based on the history-pair \( \langle l, H_l \rangle \), it checks its authenticity against authentication \( A = [R_l] \). Let \( \langle V_c, H_c \rangle \) be the local history-pair of \( c \) that the server associated with the above response. The client checks that \( \langle V_c, H_c \rangle \) is authentic with respect to \( A \) and hence is on the same history of \( \langle l, H_l \rangle \). If \( V_c \leq l \), than the server declared a version of \( D \) which is equal or after that identified by \( V_c \), respecting monotonicity. Analogous reasoning can be done for authenticators. They additionally authenticate a new version of \( \Pi \) with the new history-pair, but only if the monotonicity checks were successful. \( \square \)

**Lemma 2** (Commit-Correctness). *In an execution of the history-integrity protocol in which no trusted entity detects any tampering, the sequence of events seen by each client \( c \) is commit-correct.*

*Proof.* Let \( \sigma \) be the sequence of the events of an execution of the history integrity protocol in which no trusted entity detects any tampering. Let \( \pi_c \) be the complete sequential permutation of the subsequence of \( \sigma \) seen by \( c \). In the history-integrity protocol the evolution of data occurs at each commit. Each commit monotonically grows the history of root-hashes (see Property 4) currently seen by authenticators. We denote by \( \chi_i \) the commits seen by \( c \).

We note that the alternating structure of \( \pi_c \) mandated by Definition 1 is implied by the fact that commits deal only with updates and are atomic. Then, to prove that \( \pi_c \) is commit-correct, we have to prove that the three conditions of Definition 1 holds for the sequence \( \chi_i \).

Conditions 1 and 2 are verified since, each read has in its response the indication of the associated version of the history-hash. Further, \( c \) performs verification, by checking proofs, that the returned value of the read is indeed associated with the current version declared by the server in the response or it is initial.

Concerning Condition 3 at each commit, the server proposes to the authenticator a sequence of updates. The authenticator checks that their order
conforms to that specified by each client (by hash chaining) and that the server does not propose already processed updates (by checking past-hashes). Since in each authentication round the authenticator deals only with updates that have to be associated with the current commit, Condition 3 is verified.

**Theorem 2** (Quasi-fork-linearisability for the history-integrity protocol). The history-integrity protocol emulates a key-value store on a Byzantine server with quasi-fork-linearisability in monotonic sense.

*Proof.* We consider a generic execution of the history-integrity protocol, in which no trusted entity detects any tampering. The execution is represented by a real-time ordered sequence of events $\sigma$.

Recalling Definition 5, we should prove that, for all client $c_i$, the corresponding $\sigma_i$ and $\pi_i$, chosen by the server, satisfy the four conditions of quasi-fork-linearisability and that each $\pi_i$ grows monotonically over time. Monotonicity is stated by Property 4.

About Condition 1, client $c_i$ receives $[R_l]$, signed by an authenticator, which also authenticates $H_l$. Hash $H_l$ is uniquely associated with the sequence of updates seen by $c_i$ which is $\pi_i$. The protocol mandates that the authenticator, which receives from the server proofs based on $[R_j]$, creates $[R_{j+1}]$ by adding, among all the others, all updates of $c_i$ communicated by the server. The server can not skip or reorder any of them since they are hash-chained, and can not go back in time since past-hashes are checked. Hence, the authenticator provide a proof that they are contained in $\pi_i$. This is true for all authentication rounds. Since no trusted entity detects any tampering, all completed updates of $c_i$ are contained in the last version of the dataset $D_l$ seen by $c_i$. This proves Condition 1.

About Condition 2, by Lemma 2, all $\pi_i$ are commit-correct. By Lemma 1, they preserve the real-time order of all non-commuting operations in $\sigma$.

About Condition 3, for the updates this condition is enforced by an authenticator when it checks proofs and computes the new $[R_l]$ for the new $D_l$. For the read operations, this condition is enforced by the checks performed by each client $c_i$. Clients check that the read result comes from $D_l$ and that $D_l$ is an updated version of the last version seen by $c_i$.

About Condition 4, if the server does not introduce any fork, this condition is trivially verified by $\sigma = \sigma_1 = \sigma_2$, $\pi_1 = \pi_2$. Let assume that the server does fork, and $\pi_1 = \alpha_1 \beta_1$ and $\pi_2 = \alpha_2 \beta_2$ be the sequential sequences seen by clients $c_1$ and $c_2$ as in Definition 4.
Without loss of generality, we consider the point of view of \( c_1 \). We now prove that all update invocations of \( c_1 \) in \( \beta_1 \cap \beta_2 \) are before the first response to \( c_1 \) in \( \beta_1 \cup \beta_2 \), in \( \sigma \). By contradiction, we suppose that this is not true and prove that a tampering must be detected by a trusted entity. Let \( u \) be an update whose invocation is in \( \beta_1 \cap \beta_2 \) and, against Condition 4, let \( o \) be the last operation whose response is in \( \beta_1 \cup \beta_2 \) and \( \text{res}(o) < \text{inv}(u) \). When \( c_1 \) receives \( \text{res}(o) \), the history-pair associated with the version of the dataset on which \( \text{res}(o) \) is based is pushed into local history-pairs \( \Gamma \) of \( c_1 \). By the way history-pairs are built, the last inserted element of \( \Gamma \) of \( c_1 \) uniquely identify one of the two branches, since \( \beta_1 \neq \beta_2 \) by definition. The server is showing to \( c_1 \) operations according to \( \pi_1 \), hence, at this point, the tail \( p \) of \( \Gamma \) is associated with \( \alpha \beta_1 \). Client \( c_1 \) prepares \( \text{inv}(u) \) specifying \( p \) as history-pair and signing it with the whole \( u \). When the server prepares the authentication request containing \( u \), it cannot change \( p \) embedded in \( u \). Since \( u \) is in both branches of the fork, two authentication requests containing \( u \), one for each branch, are sent by the server to (possibly distinct) authenticators. The processing of the authentication request associated with \( \alpha \beta_1 \) completes successfully since \( p \) is proven to be authentic with respect to the history root-hash provided by the server. The processing of the authentication request associated with \( \alpha \beta_2 \) fails when trying to authenticate \( p \) in \( u \) against the history root-hash associated with \( \alpha \beta_2 \) which is not the one seen by \( c_1 \). Hence, the tampering is detected.

Note that in the above description authenticators are not linked to a branch. In fact, since they are stateless, they have no mean to detect they are used by the server to authenticate commits for distinct branches. This does not have any impact on quasi-fork-linearisability from the point of view of the clients. However, nothing prevents to equip authenticator with a state similar to that of the clients to allow them to perform similar checks. In this way, the number of branches that the server could possibly create would be bounded by the number of available authenticators.

**Corollary 1 (No False Negatives for the history-integrity protocol).** In the history-integrity protocol, whenever a trusted entity detect a tampering, the server deviated from the quasi-fork-linearisability behaviour.

Corollary 1 directly derives from Theorem 2.
**Theorem 3** (No false positives for the history-integrity protocol). *In the history-integrity protocol, whenever the server behaves according to quasi-fork-linearisability, trusted entities do not detect any tampering.*

*Proof.* We assume correct implementation of ADSes and cryptographic primitives. Tampering is detected by trusted entities when one of the checks they perform fails. All trusted entities checks that the version declared in each message form the server is in the same history of their locally stored history-pair (monotonicity). Clients check correctness of replies against last root-hash authentication (correct operation execution). Authenticators check that for each client $c$ each $u^c(i)$ is correctly hash-chained to the one before or with current past-hash $\eta^c$ and check the authenticity of $\eta^c$ for all $c$ (server does not reorder updates). Authenticators check the authenticity of previous values $v$ of $k$ (correct operation execution). Authenticators check the history-pair $p^c$ contained in the last $u^c(i)$ specified in the sequence for $c$ is in the history of the current history root-hash (Condition 4 of Definition 5). All these checks are successful if server and all authenticators behaved correctly till that moment, which is true by hypotheses. \(\square\)

### 8.3. Limiting the storage needed by server and clients

According to the above description, the server should store $\Pi$, each client should store its $\Gamma$ and these data structures grow over time. For datasets that last long and change frequently, this is an overwhelming burden. We now show how to bound the storage taken by $\Pi$ and $\Gamma$.  

---

**Figure 11:** Relationship between messages and changes applied to $P(c)$ and $\Gamma$. 

---
The server keeps a mapping $P$ from each client $c$ to a queue of history-pairs. We denote $P(c)$ the queue associated with $c$. An history-pair $p = (l, H_l)$ is pushed into $P(c)$ when a response to $c$ is sent with the version $l$, unless $P(c).\text{tail}$ is already equal to $p$. Hence, in queues $P(c)$, history-pairs are stored, from head to tail, in ascending order of version. A client $c$ that receives a response containing $p$ pushes it into $\Gamma$, unless $\Gamma.\text{tail}$ is already equal to $p$. Hence, also $\Gamma$ stores history-pairs in ascending order of version. When $c$ sends an update invocation, it uses $p = \Gamma.\text{tail}$ as history-pair in the construction of the update. When $c$ sends a read invocation, it additionally specifies $p$ in the message. When the server receives a read invocation with $p$, or puts an update containing $p$ into an authentication request, it pulls from $P(c)$ all history-pairs with a version less then that of $p$, so that $P(c).\text{head} = p$. When the server sends a response to $c$, it adds $P(c).\text{head}$ as acknowledgement of its reception, piggybacked. When $c$ receives a response that acknowledges $p$, it pulls from $\Gamma$ all history-pairs with a version less then that of $p$, so that $\Gamma.\text{head} = p$. The way messages affect $P(c)$ and $\Gamma$ is summarised in Figure 11.

The server exploits the pruning feature of ADSes (see Section 3), keeping in $\Pi$ only history-pairs that are mentioned at least once in any queue $P(c)$ for any $c$ (see Figure 12). Note that, pruning does not change the current history root-hash, it just reduces the memory occupation.

This scheme ensures that (i) each time a client $c$ uses $p$ as history-pair in an update, the server can provide proof($\Pi, p$) since $p$ is not pruned, (ii) each time $c$ receives a response with acknowledge $p$, $c$ has $p$ in $\Gamma$ (if the server has not forked) hence checks for monotonicity and quasi-fork-linearisability work, (iii) the size of each queue $P(c)$ is bounded by the number of commits involving updates from $c$ that are sent in the time of serving one update, and (iv) the same bound holds for $\Gamma$.

Pruning does not make proofs shorter, this means that their length is $O(\log l)$, for a typical ADS, which might be not acceptable for datasets that are updated regularly and must last long. For simplicity of explanation, we suppose $\Pi$ is realised with a binary MHT. If a client $c$ regularly performs queries, the version $V^c$ of $P(c).\text{head}$ is close to $l$. If this is true for all clients, the left subtree of the root in $\Pi$ is completely pruned and we can substitute its current root with its right child shortening the length of the proofs. While this makes the implementation of server and authenticators a bit more complicated, it allows us to have the proofs derived from $\Pi$ of size $O(\log(l - \min_c V^c))$. Size of the proofs can be kept bounded if we “detach” clients that are stale for a number of commits greater than a fixed threshold.
Figure 12: An example of pruning. Dots represent unpruned versions, while crosses represents pruned ones. Pruning of a history-pair from \( H \) occurs when no \( P(c) \) contains it.

9. The Pipeline-Integrity Protocol

This section presents a protocol, which we call pipeline-integrity protocol, that is scalable and achieves quasi-fork-linearisability. Essentially, we prove that results shown in Section 7 and those shown in Section 8 can be combined. We just describe the specificities of the use of the two approaches together. The complete pseudocode for the resulting pipeline-integrity protocol is provided in the Appendix.

In the following description of the protocol, we reuse many concepts and assumptions introduced in the simplified version (Section 7). Namely, we assume

- to have \( q \) authenticators,
- to have negligible execution time on server and authenticators,
- to have reliable and synchronous communications,
- to send an authentication request to authenticator \( a \) when an authentication reply is received from \( a \), and by synchronous operation this occur every \( \Delta t = 2d/q \),
- to have a pipeline-like interaction scheme between server and authenticators, and
• to have a server with U/R-data-structures, where an R-data-structure tracks the corresponding U-data-structure with a delay of $q\Delta t$.

We also reuse the same notation introduced in Section 7 to distinguish U/R-data-structures, with superscripts $^R$ and $^U$, and for denoting instances of server variables between instants $t_j$ and $t_{j+1}$, like $D_j^U$.

From the history-integrity protocol (Section 8), we reuse

- the concepts of history-hash $H_j = \text{hash}(H_{j-1}\|r_j)$ and history-pairs $(j, H_j)$,
- the fact that each client $c$ keeps a queue $\Gamma$ of local history-pairs that it is aware of.
- the content of the update operations and their notation: $u^c(i) = (k_i, v_i, \text{hash}(u(i-1)), i, (V^c, H^c))$
- the use of the mapping $P(c)$ to track, on the server, the last history-pairs sent by the server to each client $c$,
- the use of pruned authenticated data structures on the sequence of history-pairs that, for the pipeline-integrity protocol, are two: $\Pi^R$ and $\Pi^U$, and
- the notation for history root-hash $R_j$, that we use to denote the root-hash of $\Pi^U$ between instants $t_j$ and $t_{j+1}$.

While we refer the reader to the proper section for an explanation of the above concepts, we now explicitly describe the parts that need specific explanation.

**Status.** Table 4 summarises the variables that form the status of the server. Many variables are the same as in the simplified version, in particular $l$ is the index of the last instant in which the server received an authentication reply (and sent an authentication request). Consider the sequence of history pairs $\langle 0, H_0 \rangle, \ldots, \langle l, H_l \rangle$. The server keeps two history ADS, denoted $\Pi^R$ and $\Pi^U$, on this sequence. Structure $\Pi^U$ is over the whole sequence and its history root-hash is denoted by $R_l$. Structure $\Pi^U$ is used with $D^U$ and $\Delta^U$ to compute proofs for a new authentication request $\rho_l$ to be sent at instant $t_l$. Structure $\Pi^R$ is limited up to index $l - q$ and its history root-hash is $R_{l-q}$,
Figure 13: A scheme showing the data structures involved in the (complete) pipeline-integrity protocol.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D^R, D^U, \Delta^R$, $\Delta^U$, $l$, $Q^c$, $\Omega$</td>
<td>See Table 1.</td>
</tr>
<tr>
<td>$\Pi^U$</td>
<td>ADS on the sequence of history-pairs up to $\langle l, H_i \rangle$, used in conjunction with $D^U$ and $\Delta^U$. The root-hash of $\Pi^U_j$ is denoted $R_j$.</td>
</tr>
<tr>
<td>$\Pi^R$</td>
<td>ADS on the sequence of history-pairs up to $\langle l-q, H_{l-q} \rangle$ used in conjunction with $D^R$ and $\Delta^R$. Its root-hash is $R_{l-q}$.</td>
</tr>
<tr>
<td>$P$</td>
<td>A mapping from each client $c$ to a queue of history-pairs. The queue associated to $c$ is denoted $P(c)$.</td>
</tr>
<tr>
<td>$A$</td>
<td>Authentication for $D^R$, $\Delta^R$ and $\Pi^R$ in the form $[R_{l-2q}, R_{l-2q+1}] [R_{l-q-2}, R_{l-q-1}] \ldots [R_{l-q-1}, R_{l-q}]$.</td>
</tr>
<tr>
<td>$\bar{A}$</td>
<td>Sequence of the root-hashes $R_{l-2q}$, $R_{l-2q+1}$, $\ldots$, $R_{l-q}$ that are the basis of $A$.</td>
</tr>
</tbody>
</table>

Table 4: Status of the server for the (complete) pipeline-integrity protocol.
since $\Pi^R_l = \Pi^U_{l-q}$. Structure $\Pi^R$ is used with $D^R$ and $\Delta^R$ to reply to read requests. Figure 13 pictorially shows the relationships among all the data structures. Structures $\Pi^R$ and $\Pi^U$ are kept pruned by using the mapping $P$ as explained in Section 8.

The server keeps a chained authentication $A$ for the history root-hash $R_{l-q}$ in this form: $[R_{l-2q}] [R_{l-2q}, R_{l-2q+1}] [R_{l-2q+1}, R_{l-2q+2}] \ldots [R_{l-q-1}, R_{l-q}]$. The sequence $\overline{A}$ of the history root-hash involved in $A$ is kept as well.

Messages. The messages follow the scheme of the simple pipeline-integrity protocol with the following changes.

- Whenever a (signature of a) root-hash is specified for the simplified version, a corresponding (signature of a) history root-hash is specified for the complete version.

- In responses from server to clients, proofs of the kind $\text{proof}(\Delta^R, \cdot)$ in a message for the simplified version are substituted with quadruple containing $\text{proof}(\Delta^U_{l-1}, \kappa^c, \eta^c), \text{proof}(\Pi^U_{l-1}, p^c)$ where $p^c$ is the history-pair $\langle V^c, H^c \rangle$ in the last update of $c$ included in the request.

- For authentication requests, updates are represented as $u^c(i) = \langle k_i, v_i, \text{hash}(u(i-1)), i, \langle V^c, H^c \rangle \rangle$ (see Section 8). In each request to authenticator $a$, $H_{l-2}, l$ and $\text{proof}(\Pi^U_{l-1}, \langle l-1, H_{l-1} \rangle)$ are additionally sent. For each client $c$ involved in the updates in the request, the following are included: $\langle \kappa^c, \eta^c \rangle, \text{proof}(\Delta^U_{l-1}, \kappa^c, \eta^c), p^c, \text{proof}(\Pi^U_{l-1}, p^c)$ where $p^c$ is the history-pair $\langle V^c, H^c \rangle$ in the last update of $c$ included in the request.

Behaviour. Clients behave the same as in the history-integrity protocol.

The server executes both the behaviour specified for the history-integrity protocol and for the simplified pipeline-integrity protocol as follows. When handling the authentication reply $\rho^{\text{pl}}$, to a request $\rho$, from $a$, $P(a)$ is updated, $D^R$, $\Delta^R$ and $\Pi^R$ are updated according to the updates specified in $\rho$, as in the history-integrity protocol, and $A$ and $\overline{A}$ are updated, as in the simplified pipeline-integrity protocol. When creating the new authentication request $\rho'$ to be sent to $a$, the sequence of updates is created as in the history-integrity protocol, and proofs needed to create $\rho'$ are collected. Then, $D^U$, $\Delta^U$ and $\Pi^U$ are updated to be ready for the next authentication request. Finally, pruning is done on both $\Pi^R$ and $\Pi^U$ based on the content of $P$. 
Concerning authenticators, when authentication request $\rho$ is received, compaction of chained authentication $A$ contained in $\rho$ is executed as in Algorithm 2, but substituting regular root-hashes with history root-hashes. About the authentication of update operations, the performed checks are the same as in the history-integrity protocol, but executed in a conditional manner. That is, no verification is performed against the authentication $A$ communicated by the server, since this is late of $q$ instants with respect to proofs. Instead, first the conditioning history root-hash $\mathcal{R}$ is computed from an arbitrarily chosen proof in $\rho$. Then, all checks are performed against $\mathcal{R}$ to verify coherency of $\rho$. If all checks are successful, the new conditioned history root-hash $\tilde{\mathcal{R}}$ is computed on the basis of update operations in $\rho$ and new past-hashes for each client. Then, $[\mathcal{R}, \tilde{\mathcal{R}}]$ is put into $\rho^{\text{up}}$ along with the compacted version of $A$.

**Theorem 4** (Quasi-fork-linearisability for the pipeline-integrity protocol). The pipeline-integrity protocol emulates a key-value store on a Byzantine server with quasi-fork-linearisability in monotonic sense.

*Proof.* The proof of this theorem is a consequence of Theorem 2 of Properties 2 and 3 and of the following considerations.

Consider a generic execution of the pipeline-integrity protocol, in which no trusted entity detects any tampering. We call it $\mathcal{P}$. The execution is represented by a real-time ordered sequence of events $\sigma$. The server may decide to arbitrarily fork, hence, each client $c_i$ sees a subsequence of events $\sigma_i$, executed according to a sequential permutation $\pi_i$ of $\sigma_i$ and a sequence $\chi_{i,j}$ of commits seen by $c_i$. Consider $\sigma$ as an execution of the history-integrity protocol (see Section 8), that we call $\mathcal{H}$ and use Theorem 2 with the same choices of $\sigma_i$, and $\pi_i$ to prove quasi-fork-linearisability. In $\mathcal{H}$, commits $\chi_{i,j}$ are instantaneous and are in one-to-one correspondence with the commits of $\mathcal{P}$. Association between operations and commits in $\mathcal{H}$ also mimic what happen in $\mathcal{P}$. In $\mathcal{H}$, all update operations are executed not before $q$ commits.

This might sound weird for a real non-pipelined system, but it conforms to the interaction scheme introduced in Section 4.3 and it is compatible with the history-integrity protocol. Hence, Theorem 2 applies. Note that, the checks performed by trusted entities in $\mathcal{H}$ and $\mathcal{P}$ are the same, with the only difference that in $\mathcal{P}$ proofs are verified against chained authentications. The second notable difference is that authenticators in $\mathcal{P}$ provide a conditional authentication not a plain one. However, in our synchronous and reliable execution, after $q$ instants, the chain is complete and, by Properties 2 and 3.

57
what was conditionally verified \( q \) instants before is verified unconditionally at current instant. \( \square \)

Corollary 2 directly derives from Theorem 4.

**Corollary 2** (No false negatives for the pipeline-integrity protocol). *In the pipeline-integrity protocol, whenever a trusted entity detect a tampering, the server deviated from the quasi-fork-linearisability behaviour.*

**Theorem 5** (No false positives for the pipeline-integrity protocol). *In the pipeline-integrity protocol, whenever the server behaves according to quasi-fork-linearisability, trusted entities do not detect any tampering.*

**Proof.** The proof of this theorem is a consequence of Theorem 3 of Properties 2 and 3 and of the following considerations.

We assume correct implementation of ADSes and cryptographic primitives. Tampering is detected by trusted entities when one of their checks fails. Each check that is performed by trusted entities in the pipeline-integrity protocol matches one in the history integrity protocol, with the exception of the checks introduced by the verification of the coherency of the chained authentication. The following differences should be considered. In the pipeline-integrity protocol, clients perform their checks against a chained authentication of the history root-hash instead of against a plain authentication. The semantic equivalence of the two approaches is stated by Properties 2 and 3 and, for clients, the statement holds by Theorem 3. Authenticators do not perform real checks, but assume that a certain history root-hash \( \bar{R} \) is authenticated and perform their checks against it. Then, they state the authenticity of the new history root-hash \( \tilde{R} \), after updates application, conditioned to the authenticity of \( \bar{R} \). These conditioned authentication will form a chained authentication whose coherency is checked by trusted entities (see Algorithm 1) when they are received from the server. All these checks are successful if server and all authenticators behaved correctly till that moment, which is true by hypotheses. \( \square \)

Concerning scalability, we can follow the same reasoning of the scalability analysis shown in Section 7, considering non-negligible computation and transmission time. The following holds.

**Theorem 6** (Scalability). *In the pipeline-integrity protocol, there exists a value of the number of authenticators \( q \) for which the response time for each*
authentication request is bounded by $4\beta\alpha\lambda + 2\gamma$ for $\lambda$ large enough, where $
abla$ is the total processing/transmission time for one update, $\beta$ is the total processing/transmission time for one conditional authentication, and $\gamma$ is the remaining processing/transmission time and network delay in a round that does not depend on the number of updates in the request or $q$.

Proof. The statement is a direct consequence of Theorem 1 and of the fact that the pipeline-integrity protocol follows the same interaction scheme of the simplified pipeline-integrity protocol.

10. Dealing with Non-Ideal Resources

In practice, assuming that the pipeline-integrity protocol can run synchronously is unrealistic for most applications, because real networks may suffer packet losses and congestion. Further, performances of machines may vary depending on other processes. First, we observe that we can adopt a reliable transport protocol, like TCP, that implements acknowledgements and retransmissions. It allows the two parties to send an amount of non-acknowledged data that is enough to saturate the (estimated) bandwidth of the channel. This permits us to deal only with uncertainty about network delays or with the (un)availability of an authenticator. For a practical realization of the pipeline-integrity protocol, we suggest to decouple sending of authentication requests from the reception of authentication replies. The easiest approach is to send authentication requests at scheduled instants or when the number of accumulated updates reaches a certain threshold, but other policies might be adopted to reduce response time. At the receiving of an authentication reply, if this was the first outstanding request in the $\Omega$ queue, the regular processing is performed. If this is not the case, an intermediate authentication response was missed. This might be late or the corresponding authenticator might be unavailable or unreachable. We can wait a timeout and possibly re-send the same request to another authenticator. During the handling of a missing request, the read operations are served against the last $D^R$ and processing of updates goes ahead, but authentication requests and replies should be buffered till the missing piece will arrive. Application specific considerations should lead the decision on which architecture elements should play the role of authenticators. This choice should depend not only on their computational power but also on the probability to become unreachable and on the fact that this is not a problem if at the
same time also clients get disconnected. To reduce the probability of missing an authentication reply, more than one authenticator might be involved for each authentication round. A flow control should also be realised so that the server could communicate to clients that it cannot go ahead with updates (or can only at a slow pace) due to non responding authenticators. In this way, the clients can limit or stop updates and possibly informing the user of the problem.

In a real environment, clients may be abruptly disconnected or switched off. In this case, some response messages may be lost, even when a reliable transport protocol like TCP is used. In general, when the client reconnects, the server does not know which is the last version it is aware of since it does not know which is the last response messages the client processed. However, the protocol described in Section 8.3 is tolerant with respect to this problem, provided that client can keep content of $\Gamma$ across reconnection, for example, by persisting it.

11. Experimental Study

We developed a prototype with the intent to provide experimental evidence of the feasibility of our approach in a realistic setting. Since our contribution provides means to scale the trusted side, we designed our experiments so that the bottleneck of the whole system is always the computing power dedicated to authenticators. We note that the average latency (i.e., the time taken for an update request to be authenticated) is proportional to the duration of one authentication round. We measured the duration of one authentication round and other parameters at increasing update invocation rate. We measured the maximum update invocation rate of that the system can sustain (i.e., its throughput) with different numbers of CPUs dedicated to run authenticators.

We note that authenticators may be implemented, as processes, threads or actors (see for example the Akka actors [18]) which essentially allow us to increase their number without substantial additional cost. Clearly, their overall speed is bounded by the number of available physical processors. We denote by $q$ the number of physical processors dedicated to execute authenticators, by $t_A$ the time each of them takes to process one update (ignoring all other contributions for simplicity), and by $\lambda$ the update arrival frequency. For a well dimensioned system it should be $ar{q} \geq \lceil mt_A/\Delta t \rceil = \lceil \lambda t_A \rceil$ (see Figure 14). In this way, when a new authentication request arrives, there is
always a free processor for it. Hence we expect a linear relationship between throughput and number of CPUs. The objectives of our experiments are the following.

O1. We intend to show that our approach supports update frequencies, up to the saturation of the CPUs of the authenticators, keeping the response time practically constant.

O2. We intend to show that it is possible to increase the throughput of the system by increasing the number of processors dedicated to execute authenticators.

O3. We intend to show that it is possible to obtain a minimum response time, as described in Section 7.5, by tuning the number of updates contained in each authentication request.

Our experiments focus on update operations, since read operations are only marginally affected by the introduction of the pipeline-integrity protocol. Our prototype is explicitly targeted to this experimentation. Providing a fully fledged implementation was not among our objectives. We implemented the simplified pipeline-integrity protocol described in Section 7, supporting read operations and update operations only for keys already existing in the dataset. We note that the scheme of client-server interaction and server-authenticator interaction in the simplified and in the complete versions of

Figure 14: Construction to prove that $\bar{q} \geq \lceil mt_A/\Delta t \rceil = \lceil \lambda t_A \rceil$ is enough to ensure that each request has a free processor for it. In the example $\bar{q} = 4$. 

\[ \bar{q} \cdot \Delta t \text{ where } \bar{q} = \lceil mt_A/\Delta t \rceil \]
the protocol is the same. The most notable difference between the two versions of the protocol is that in the simplified version no history ADS is kept, so proofs do not encompass the integrity of the history. This means that the number of cryptographic hashes that all machines have to compute is less than in the case of the complete version (see Section 8) and the length of messages is correspondingly shorter. However, regarding the management, transmission and check of the chained authentication, which is the core of our approach, there is no relevant difference.

We performed our tests with three machines: \( S \) running our server, \( C \) running a number of clients that perform update requests to \( S \) and \( A \) running authenticators to serve authentication requests sent by \( S \). All machines are Linux-based. Machines \( A \) and \( C \) were located at the Computer Network Research Laboratory at Roma Tre University. Machine \( A \) was an old machine with 4 CPUs at 2.4GHz. Machine \( S \) was a large server (64 CPUs) located in a cloud that is operated by Consortium GARR, which is also the connectivity provider of the Roma Tre University. Round-trip delay between \( S \) and \( A \) was about 20ms. The bandwidth between the laboratory and \( S \) is large enough to rule out any congestion, however, due to an old 100Mbit ethernet switch near to \( A \), the bandwidth between \( S \) and \( A \) turned out to be 10.5Mbyte/sec\(^2\). Since this bandwidth is quite small, to avoid the risk for the network to be the bottleneck of our experiments, we artificially increased the CPU consumption on \( A \) of each authentication round by performing additional dummy cryptographic operations. In all tests, we also checked that the server was not a bottleneck\(^3\). Section 10 deals with a number of non ideal aspects of real systems. Most of them turned out to be irrelevant in the controlled environment of our experiments. Our software is based on the Akka library realizing the actor model \[18\]. The communication among machines takes advantage of the Akka proprietary protocol (TCP-based). Since this protocol is designed to work in LAN, we set up a tunnel between our laboratory and \( S \). Our software runs on the Java Virtual Machine. Since the JVM incrementally compiles the code according to the “HotSpot” approach \[32\], for each run we took care to let the system run long enough before performing the measurements, to be sure that compilation activity was over. Concerning

---

\(^2\)The measurement was performed by the standard iperf tool ([https://iperf.fr/](https://iperf.fr/)).

\(^3\)The measurement of the consumption of system resources was performed by the standard dstat tool ([http://dag.wiee.rs/home-made/dstat/](http://dag.wiee.rs/home-made/dstat/)).

\(^4\)The tunnel was realized by the socat tool ([http://www.dest-unreach.org/socat/](http://www.dest-unreach.org/socat/)).
the ADS, we realized an Authenticated Skip list containing 100000 keys [17]. This structure is created and randomly initialised at the start up of the system.

We first measured the performances of the blocking approach (see Section 5), which we realised using the same software imposing $q = 1$, where $q$ is the number of outstanding authentication requests. Due to this constraint, in this case, we always wait for the arrival of the authentication reply and send, in the next request, all updates arrived in the meantime, which is larger than 200, much like in the description given in Section 5. Figure 15 shows how several parameters changes ($y$-axes) when the frequency of updates arrivals ($x$-axis) increases. We observe that near 325 upd/sec the round duration increases steeply, and above that frequency, the messages grow so big that they can hardly be handled without errors. Hence, the throughput of this setting is about 325 upd/sec.

To meet Objectives O1 and O2 we measured the performance of the simplified pipeline-integrity protocol described in Section 7 with an increas-
The results are shown in Figures 16, 17, 18 and 19. The x-axis shows the update invocation rate. The y-axes show the authentication round duration, the percentage of the CPU used by Machine A (400% means that all 4 CPUs are fully used), the data received by Machine A, and the average value for the outstanding authentication requests $q$. The tests show that the round duration is largely constant until the allocated CPUs are close to saturation (see Objective O1) and when $q$ increases, as in Figure 19, even slightly decreases. We note that, adding CPUs increases the throughput but has no detrimental effect on latency, which was the scalability objective.

---

Figure 16: Performance of the simplified pipeline-integrity protocol when authenticators are limited to use 1 CPU. Throughput is between 350 and 400 upd/sec. Latency is 622–662msec.

The limit on the number of CPUs used is imposed by using the CPU affinity setting provided by Linux.
that was stated in Section 4.2. We also note that, for one CPU, the throughput is similar to the one for the blocking approach. In addition, we obtain the advantage that, for update rate close to the throughput, adopting the pipelining approach allows the (average) value of $q$ to increase making the system much more stable than for the blocking approach. For each experiment, we visually identified the consecutive frequencies close to the frequency where CPUs of machine $A$ saturate. We considered this pair of frequencies an approximation of the throughput of the system. Figure 20 summarises the measurements of the throughput of all four experiments. The x-axis shows the number of CPUs of Machine $A$ that authenticators are allowed to use. The y-axis shows the throughput of the system deduced from the inspection of Figures 16, 17, 18 and 19. The chart shows that the throughput linearly increases with the number of CPUs used by the authenticator, as expected (see Objective O2).

To meet Objective O3 we performed an experiment to show the relationship between $m$, $q$ and the round duration $T$. Since in our software $q$ is automatically adjusted, to increase $q$ we decreased $m$ and computed an
The average value of $q$ as $(\lambda T)/m$. In this experiment, we removed all additional dummy cryptographic operations introduced for the previous experiments and fixed $\lambda = 500$. This value ensured that we never reached any bottleneck of the system during the test. In this experiment, we allowed all four CPUs to work. The result is depicted in Figure 21. The x-axis shows the average number of outstanding requests $q$. The y-axis shows the duration of the authentication round. Each point is labelled with the average value of updates in one authentication request. The chart shows that the round duration decrease until $m = 35$, where $q_{\text{min}} \approx 2.21$ and $T_{\text{min}} \approx 154.97$. After $q_{\text{min}}$, $T$ rises with small slope as predicted by the theory (see Section 7.5).
Figure 19: Performance of the simplified pipeline-integrity protocol when authenticators are limited to use 4 CPUs. Throughput is between 1700 and 1750 upd/sec. Latency is 588–616 msec.

Figure 20: Relationship between the number of CPUs available for the authenticators and throughput of the system.
12. Conclusions

We have presented the pipeline-integrity protocol, which enables the use of authenticated data structures to check integrity of outsourced key-value stores at untrusted servers in a setting in which many clients concurrently perform updates. The main features of our methodology are its scalability and its ability to detect server misbehaviour in the quasi-fork-linearisability consistency model. This consistency model is almost as strong as the one which was theoretically proven to be the strongest possible in that setting (fork-linearisability). Our approach is scalable in the sense that, it is possible to saturate the system bottleneck without substantial increase of the response time. We also support concurrent authentications, which makes much easier to solve bottlenecks on the trusted side. One notable aspect of our result is that scalability holds even when the round-trip delay between server and authenticators is large and update rates are high, which were commonly regarded as bad situations for adopting authenticated data structures. We proved practical performance with an experimental evaluation.

We think that the pipeline-integrity protocol may help the adoption of authenticated data structures in many contexts. In particular, in those where a
large number of small devices may need to access and update integrity-critical
data, like, cloud-based mobile applications, Internet of Things, industrial sys-
tems (Industry 4.0), financial systems, and micropayments systems.

There are several future research directions suggested by our work. On the
theoretical side, one can ask if the quasi-fork-linearisability is the strongest
feasible consistency model that scales, in the sense adopted in this paper.
Regarding the adoption in cloud facilities, we note that many of the tech-
nologies that underlie the cloud favour availability vs. consistency, according
to the CAP/PACELC theorem [1]. Since our approach detects consistency
violations, it does not fit well with these technologies. This essentially limits
the use of the pipelining-integrity protocol to situations where partitioning is
ruled out by proper network configurations. On this side, one may ask if the
detection of a fork may be recovered gracefully by authenticators. They may
recognise the join as not harmful in many non malicious cases and resort to a
human-based fix when there is no safe automatic merge approach. Further,
one can ask if the techniques described in this paper can be adopted in con-
junction with a decentralised P2P network (like, for example, that described
in [19]), where the P2P network is considered an untrusted storage. In this
context, all nodes have to agree on the same sequence of updates in order to
consistently reply to read invocations.

Finally, it should be nice to have a publicly available library implementing
the pipelining-integrity protocol to smooth its adoption into real applications.

13. Acknowledgements

We would like to thank Consortium GARR for providing us with the cloud
resources we used for our experiments. We would like to thank Gianmaria
Del Monte and Matteo Giordano for their help in writing a first version of
the software we used for the experiments.

This research did not receive any specific grant from funding agencies in
the public, commercial, or not-for-profit sectors.

14. References

system design: Cap is only part of the story. Computer, 45(2):37–42,
2012.


[29] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. HASP@ ISCA, 10, 2013.


Appendix. Formal Description of the Pipeline-Integrity Protocol

Notation

For a detailed description, please refer to Sections 8 and 9.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>A key in a dataset.</td>
</tr>
<tr>
<td>$v$</td>
<td>Value associated with a certain key in a dataset.</td>
</tr>
<tr>
<td>$\langle k, v \rangle$</td>
<td>Key-value pair in a dataset.</td>
</tr>
<tr>
<td>$\kappa^c$</td>
<td>A client-key for client $c$.</td>
</tr>
<tr>
<td>$\eta^c$</td>
<td>The past-hash associated with $\kappa^c$.</td>
</tr>
<tr>
<td>$q$</td>
<td>Number of authenticators and depth of the pipeline.</td>
</tr>
<tr>
<td>$t_i$</td>
<td>Instants at which the server receive authentication reply and sends authentication requests.</td>
</tr>
<tr>
<td>$\rho_i, \rho_i^{\text{ret}}$</td>
<td>Authentication request (sent at instant $t_i$) and the corresponding reply (received at instant $t_i + q$).</td>
</tr>
<tr>
<td>$l$</td>
<td>Index of the last instant of time just executed by the server, which is denoted $t_l$.</td>
</tr>
<tr>
<td>$D^R, D^U$, $\Delta^R, \Delta^U$, $\Pi^R, \Pi^U$</td>
<td>Variables of the server containing readable and updated datasets, their ADS, and their history ADS. Their value changes at instants $t_i$. A subscript, like in $D_i^R$, denotes the value of that variable between $t_i$ and $t_{i+1}$.</td>
</tr>
<tr>
<td>$l, Q^c, \Omega, P, A, \overline{A}$</td>
<td>Other variables of the server.</td>
</tr>
<tr>
<td>$r_i$</td>
<td>Root-hash of $\Delta_i^U$. The root-hash of $\Delta_i^R$ is $r_{i-q}$.</td>
</tr>
<tr>
<td>$R_i$</td>
<td>History root-hash at instant $i$, that is, root-hash of $\Pi_i^U$. The history root-hash for $\Pi_i^R$ is $R_{i-q}$.</td>
</tr>
<tr>
<td>$[x]_e, [x]$</td>
<td>Signature of data $x$ performed by trusted entity $e$. If $x$ is a root-hash, it is a plain authentication of $x$. The indication of $e$ can be omitted if irrelevant.</td>
</tr>
<tr>
<td>$l, Q^c, \Omega, P, A, \overline{A}$</td>
<td>Variables on the server.</td>
</tr>
<tr>
<td>$H_i$</td>
<td>History hash associated with $D_i^U$ and $\Delta_i^U$.</td>
</tr>
<tr>
<td>$(i, H_i)$</td>
<td>History-pair related to version $i$.</td>
</tr>
<tr>
<td>$p^c = \langle V^c, H^c \rangle$</td>
<td>A history-pair known by client $c$.</td>
</tr>
<tr>
<td>proof($y, x$)</td>
<td>Proof of $x$ against an instance of an authenticated data structure $y$ (see Section 3).</td>
</tr>
<tr>
<td>$u^c(i)$</td>
<td>$i$-th update invoked by client $c$, it is a shorthand for $\langle i^c, k_i^c, v^c_i, \text{hash}(u^c(i-1)), p^c \rangle$.</td>
</tr>
</tbody>
</table>

Operations on queues

Standard semantic of queues is adopted with the following terminology. After pushing $x$ into a queue $\Gamma$, $\Gamma$.tail = $x$. If $\Gamma$.head = $x$, pulling from $\Gamma$ returns $x$. |
Algorithm 5 Variables kept by the server to store its state.

**l:** Index of the last instant in which the server performed a processing of authentication reply and sent an authentication request. That instant is denoted \( t_l \).

**\( D^R \):** The current readable dataset. This dataset is authenticated by \( A \). It holds that \( D^R = D^R_l = D^U_{l-q} \).

**\( D^U \):** The dataset that is up-to-date with respect to update invocations for which an updated request was already sent. This dataset is not authenticated. It holds that \( D^U = D^U_l = D^R_{l+q} \).

**\( Q^c \):** For each client \( c \), a queue of updates with their signatures, that is, of pairs \( \langle u^c(i), [u^c(i)]_c \rangle \) where \( u^c(i) = \langle i, k^e_i, u^e_i, \text{hash}(u^e(i-1)), p^e_c \rangle \). It contains updates that are waiting to be put into an authentication request.

**\( \Omega \):** A queue, of length at most \( q \), of outstanding authentication requests: \( \rho_{l-1}, \ldots, \rho_{l-q} \) (from last to first).

**\( P \):** A mapping from each client \( c \) to a queue of history-pairs. The queue associated to \( c \) is denoted \( P(c) \).

**\( \Delta^R \):** The ADS related to \( D^R \). It holds that \( \Delta^R = \Delta^R_l = \Delta^U_{l-q} \).

**\( \Delta^U \):** The ADS related to \( D^U \). It holds that \( \Delta^U = \Delta^U_l = \Delta^R_{l+q} \).

**\( \Pi^U \):** History ADS that contains history-pairs \( \langle i, H_i \rangle \), with \( i \leq l \), ordered by \( i \). It holds that \( \Pi^U = \Pi^U_l = \Pi^R_{l+q} \). Its root-hash is denoted \( R_l \). It is stored pruned as described in Section [9].

**\( \Pi^R \):** History ADS that contains history-pairs \( \langle i, H_i \rangle \), where \( i \leq l - q \), ordered by \( i \). It holds that \( \Pi^R = \Pi^R_l = \Pi^U_{l-q} \). Its root-hash is denoted \( R_{l-q} \). It is stored pruned as described in Section [9].

**\( A \):** The chained authentication for the current version of \( \Pi^R \) in the form \( [R_{l-2q}] [R_{l-2q}, R_{l-2q+1}] \ldots [R_{l-q-1}, R_{l-q}] \).

**\( A^{-} \):** The sequence of history root-hashes on which \( A \) is based: \( R_{l-2q}, R_{l-2q+1}, R_{l-2q+2}, \ldots, R_{l-q-1}, R_{l-q} \).

**Input:** A version index $V$ and the identifier $c$ of a client.

1: while $p = P(c).\text{head}$ has a version less than $V$ do
2: Pull $p$ from $P(c)$.
3: if version of $p$ is not referred in $P$ (for any client) then
4: Prune from $\Pi^R$ and $\Pi^U$ the entry for $p$ and all unneeded parts.
5: end if
6: end while

Algorithm 7: Server behaviour. Processing of read and update invocations.

1: upon receiving an update invocation from client $c$ containing $u^c(i)$ and its signature $[u^c(i)]_c$ do
2: Push $\langle u^c(i), [u^c(i)]_c \rangle$ into $Q^c$
3: upon receiving a read invocation from client $c$ containing the key $k$ to be read and $\langle V^c, H^c \rangle$ do
4: Execute Algorithm 6 on $V^c$ and $c$ to update $P(c)$, $\Pi^R$ and $\Pi^U$.
5: send read response to $c$ containing the value $v$ for $k$ according to $D^R$
6: $l - q$ as the version index for this response
7: proof $\langle \Delta^R, \langle k, v \rangle, H_{l-q-1} \rangle$, proof $\langle \Pi^R, \langle l - q, H_{l-q} \rangle \rangle$
8: proof $\langle \Pi^R, \langle V^c, H^c \rangle \rangle$
9: $\langle V^c, H^c \rangle$
10: $A$ and $\overline{A}$, which authenticate all the above data
11: if $P(c).\text{tail} \neq \langle l - q, H_{l-q} \rangle$ then
12: Push $\langle l - q, H_{l-q} \rangle$ into $P(c)$
13: end if
Algorithm 8 Server behaviour. Reception of an authentication reply and update of R-data-structures.

1: ▷ The variable $l$, part of the state of the server, is incremented in the algorithm below in the first step. Below, the use of the symbol $l$ always conforms to the use of $l$ throughout the whole algorithm (i.e. after the increment).

2: **upon receiving** authentication reply $\rho^{rpl} = \rho^{rpl}_{l-q}$ from a containing

3: the identifier of the associated authentication request $\rho = \rho_{l-q}$

4: $[R_{l-2q}]_a$

5: $[R_{l-q-1}, R_{l-q}]_a$

6: do

7: $l \leftarrow l + 1$

8: Get $\rho = \rho_{l-q}$ from $\Omega$ deleting it from the queue.

9: Let $\rho.B$ the sequence of updates contained in $\rho$.

10: For each client $c$ involved in $\rho.B$, update value for $\langle \kappa_c, \eta_c \rangle$ in $D_R$ and $\Delta_R$, so that $\eta_c = \text{hash}(u_c(i_{\text{max}}))$ where $i_{\text{max}}$ is the index of the last update of $c$ in $\rho.B$.

11: Apply all updates of $\rho.B$, in the specified order, to $D_R$ and $\Delta_R$.

12: ▷ Now, it holds $D_R = D^R$, $\Delta_R = \Delta_l^R$ and its root-hash is $r_{l-q}$.

13: Let $H_{l-q} = \text{hash}(H_{l-q-1}|r_{l-q})$

14: Add $\langle l - q, H_{l-q} \rangle$ to $\Pi^R$.

15: ▷ Now, it holds $\Pi^R = \Pi^R_l$ and its root-hash is $R_{l-q}$.

16: Update $A$ by substituting the two leftmost elements with $[R_{l-2q}]_a$ and appending $[R_{l-q-1}, R_{l-q}]_a$ to its right. Update $\bar{A}$ accordingly.

17: For each $c$ involved in $\rho.B$ send an update response to $c$ containing $A, \bar{A}, \text{proof}(\Pi^R, P(c).\text{head}), \text{proof}(\Pi^R, \langle l - q, H_{l-q} \rangle)$.

18: For each $c$ involved in $\rho.B$, if $P(c).\text{tail} \neq \langle l - q, H_{l-q} \rangle$ push $\langle l - q, H_{l-q} \rangle$ into $P(c)$.

19: Execute Algorithm 9 to send the authentication request $\rho_l$.

20: 78
Algorithm 9 Server behaviour. Preparation and sending of an authentication request $\rho_l$ to authenticator $a$, and update of U-data-structures.

1: $C \leftarrow$ all clients $c$ for which $Q^c$ is not empty.
2: $\rho \leftarrow$ an empty authentication request
3: \(\rho = \rho_l\). See Algorithm 10 for meaning of the fields of $\rho$.
4: $\rho.l \leftarrow l$
5: $\rho.A \leftarrow A$
6: $\rho.A \leftarrow \overline{A}$
7: $\rho.B \leftarrow$ an empty sequence
8: for all $c$ in $C$ do
9:  For all tuples in $Q^c$, add them to $\rho.B$ keeping their order.
10: make $Q^c$ empty
11: end for
12: $\rho.H \leftarrow H_{l-2}$
13: $\rho.$mainproof $\leftarrow$ proof $\left(\Pi^U, \langle l - 1, H_{l-1} \rangle\right)$ \(\triangleright\) Note that $\Pi^U$ has not been updated yet, while $l$ was, hence it holds $\Pi^U = \Pi^U_{l-1}$. The same is true for $\Delta^U$ and $D^U$.
14: for all $c$ involved in $\rho.B$ do
15:  $\rho.p^c \leftarrow p^c$ as specified in the last update of $c$ in $\rho.B$.
16:  $\rho.$histproof($c$) $\leftarrow$ proof $\left(\Pi^U, p^c\right)$, where $p^c$ is as above.
17:  $\rho.$pasthash($c$) $\leftarrow$ $\langle \kappa^c, \eta^c \rangle$ obtained from $D^U$
18:  $\rho.$pasthashproof($c$) $\leftarrow$ proof $\left(\Delta^U, \langle \kappa^c, \eta^c \rangle\right)$
19:  Execute Algorithm 6 on the version of $p^c$ and $c$ to update $P(c)$, $\Pi^R$ and $\Pi^U$.
20: end for
21: for all $k$ involved in $\rho.B$ do
22:  Let $v$ be the value of $k$ in $D^U$
23:  $\rho.$oldval($k$) $\leftarrow v$
24:  $\rho.$proof($k$) $\leftarrow$ proof $\left(\Delta^U_{l-1}, \langle k, v \rangle\right)$.
25: end for
26: send authentication request $\rho$ to $a$.
27: Push $\rho$ as last element of $\Omega$.
28: Apply all updates of $\rho.B$ to $D^U$ and $\Delta^U$ respecting their order in $\rho.B$.
29: For each $c$ with an update in $\rho.B$, update $\langle \kappa^c, \eta^c \rangle$ in $D^U$ and $\Delta^U$, where $\eta^c = \text{hash} (u^c(i_{\text{max}}))$ and $i_{\text{max}}$ the index of the last update of $c$ in $\rho.B$.
30: Let $r_l$ be the root-hash of $\Delta^U$ and $H_l = \text{hash} (H_{l-1} | r_l)$.
31: Add $\langle l, H_l \rangle$ to $\Pi^U$.
State and Behaviour of an Authenticator

Algorithm 10 Authenticator. Processing of an authentication request.
1: Let $a$ be this authenticator, and $\rho = \rho_l$ an authentication request sent by the server at instant $t_l$.

2: **state**

3: Stateless

4: 

5: **upon receiving** authentication request $\rho = \rho_l$ containing

6: identifier of $\rho$

7: $\rho.l = l$

8: $\rho.A$ = chained authentication $[R_{l-2q}][R_{l-2q}, R_{l-2q+1}] \ldots [R_{l-q-1}, R_{l-q}]$

9: $\rho.A = \text{a sequence } R_{l-2q}, R_{l-2q+1}, \ldots, R_{l-q-1}, R_{l-q}$

10: $\rho.B$ = a sequence $b_1, b_2, \ldots, b_z$ of signed updates $b_j = \left\langle u_j^{c(j)}, [u_j]_{c(j)} \right\rangle$, where $c(j)$ is the client that invoked $u_j$.

11: $\rho.H = H_{l-2}$

12: $\rho.\text{mainproof} = \text{proof } \left( \Pi^U_{l-1}, (l-1, H_{l-1}) \right)$

13: for each client $c$ involved in $\rho.B$

14: $\rho.\text{histproof}(c) = \text{proof } \left( \Pi^U_{l-1}, \rho^c \right)$

15: $\rho.\text{pasthash}(c) = \left\langle \kappa^c, \eta^c \right\rangle$

16: $\rho.\text{pasthashproof}(c) = \text{proof } \left( \Delta^U_{l-1}, \left\langle \kappa^c, \eta^c \right\rangle \right)$.

17: 

18: for each key $k$ involved in $\rho.B$

19: Let $v$ be the value of $k$ in $D^U_{l-1}$

20: $\rho.\text{oldval}(k) = v$

21: $\rho.\text{proof}(k) = \text{proof } \left( \Delta^U_{l-1}, \left\langle k, v \right\rangle \right)$.

22: 

23: **do**

24: Compact $\rho.A$ and $\rho.A$ into $[R_{l-q}]_a$ by executing Algorithm 2.

25: Compute $[R_{l-1}, R_l]_a$ by executing Algorithm 1.

26: **send** authentication reply $\rho^{\text{rpl}}_l$ containing

27: identifier of $\rho$

28: $[R_{l-q}]_a$

29: $[R_{l-1}, R_l]_a$

30: 

31:
Algorithm 11 Authenticator. Compute conditional authentication performing all related checks. In the comments, we write “should” when a certain condition is supposed to hold for executions with correctly behaving servers.

1: For each \( b_j = \langle u_j^{c(j)}, [u_j]_{c(j)} \rangle \) in \( B = \rho.B \), verify the signature of \( u_j \) in \( b_j \).

2: \( \overline{R} \leftarrow \) the root-hash computed using \( \rho.\text{histproof}(c) \) and \( \rho.p^c \), where \( c \) is an arbitrarily chosen client among those involved in \( B \).
   \( \triangleright \) It should hold that \( \overline{R} = R_{l-1} \).

3: \( \overline{r} \leftarrow \) the root-hash computed using \( \rho.\text{pasthashproof}(c) \) and \( \rho.\text{pasthash}(c) \), where \( c \) is an arbitrarily chosen client.
   \( \triangleright \) It should hold that \( \overline{r} = r_{l-1} \).

4: For each key \( k \) involved in \( B \), compute the root-hash using \( \rho.\text{proof}(k) \) and \( \rho.\text{oldval}(k) \) and verify it is equal to \( \overline{r} \).

5: for all \( c \) involved in \( B \) do

6: Verify that the updates of \( c \) in \( B \) are correctly hash-chained.

7: Verify that the hash in the first update of client \( c \) in \( B \) is equal to \( \eta^c \).

8: Verify that the root-hash computed using \( \rho.\text{pasthashproof}(c) \) and \( \rho.\text{pasthash}(c) \) is equal to \( \overline{r} \).

9: Verify that the history root-hash computed using \( \rho.\text{histproof}(c) \) and \( \rho.p^c \) is equal to \( \overline{R} \).

10: Verify that version of \( \rho.p^c \) is less than or equal to \( \rho.l - 1 \).

11: end for

12: \( \tilde{H} \leftarrow \text{hash}(\rho.\tilde{H} | \overline{r}) \).
   \( \triangleright \) It should hold that \( \tilde{H} = H_{l-1} \).

13: Verify that the history root-hash computed using \( \rho.\text{mainproof} \) and \( \langle \rho.l - 1, \tilde{H} \rangle \) is equal to \( \overline{R} \).

14: \( \tilde{r} \leftarrow \) the root-hash computed by applying all the updates \( u^c = (i, k, v, \text{hash}(u^c(i-1)), p^c) \) in \( B \) respecting their sequence in \( B \):

15: for each key \( k \) involved in \( B \), consider \( \rho.\text{proof}(k) \) and the new value \( v \) for \( k \) in the last update involving \( k \)

16: for each client-key \( \kappa^c \) for client \( c \) involved in \( B \), consider \( \rho.\text{pasthashproof}(c) \) and the new value \( \eta^c \) for \( \kappa^c \) with \( \eta^c = \text{hash}(u^c(i_{\text{max}})) \) and \( i_{\text{max}} \) as in Algorithm 9.
   \( \triangleright \) It should hold that \( \tilde{r} = r_l \).

17: \( \tilde{H} \leftarrow \text{hash}(\tilde{H} | \tilde{r}) \).
   \( \triangleright \) It should hold that \( \tilde{H} = H_l \).

18: \( \tilde{R} \leftarrow \) the history root-hash computed using \( \rho.\text{mainproof} \) and \( \langle \rho.l, \tilde{H} \rangle \).
   \( \triangleright \) It should hold that \( \tilde{R} = R_l \).

19: return \( [\overline{R}, \tilde{R}]_a \)
   \( \triangleright \) This should turn out to be \( [R_{l-1}, R_l]_a \).
Algorithm 12 Client. Reception of a read response.

1: Let $l$, $\Delta^R$, $\Pi^R$, $A$, and $\overline{A}$ denote the values of the corresponding variables of the server when the response is sent.
2: Let $c$ be this client.
3: state
4: $\Gamma$: queue of history-pairs.
5: upon receiving read response for key $k$ containing
6: $v$: the value read.
7: $l - q$: the version index of the dataset this response is based on.
8: $H_{l-q-1}$: the history-hash of the previous version index.
9: $A$ and $\overline{A}$: authentication of $R_{l-q}$
10: proof($\Delta^R, \langle k, v \rangle$)
11: proof($\Pi^R, \langle l - q, H_{l-q} \rangle$)
12: proof($\Pi^R, \langle V^c, H^c \rangle$)
13: $\langle V^c, H^c \rangle$
14: do
15: Verify $A$ and $\overline{A}$ by Algorithm 1.
16: Compute $r_{l-q}$ from proof($\Delta^R, \langle k, v \rangle$) and $\langle k, v \rangle$.
17: $H_{l-q} \leftarrow \text{hash}(H_{l-q-1} | r_{l-q})$.
18: Compute $R_{l-q}$ from proof($\Pi^R, \langle l - q, H_{l-q} \rangle$) and $\langle l - q, H_{l-q} \rangle$.
19: Verify that $R_{l-q}$ is equal to the last hash in $\overline{A}$.
20: Verify that $\langle V^c, H^c \rangle$ is in $\Gamma$.
21: Pull from $\Gamma$ all $\langle V, H \rangle \in \Gamma$ with $V < V^c$.
22: Compute $\bar{R}$ as root-hash from proof($\Pi^R, \langle V^c, H^c \rangle$) and $\langle V^c, H^c \rangle$.
23: Verify that $\bar{R} = R_{l-q}$.
24: Verify that $V^c \leq l - q$.
25: Push $\langle l - q, H_{l-q} \rangle$ into $\Gamma$.
26: done
Algorithm 13 Client. Reception of an update response.

1: > The behaviour of the client when the response to an update $u(i) = (i, k_i, v_i, \text{hash}(u(i - 1)), p)$ is received is the same as in Algorithm 12. Note that, the value returned by the server can differ from $v_i$. This occurs when, in the same commit, a distinct update $\bar{u}$ for the same key $k_i$ and value different from $v_i$ is after $u(i)$ in $B$ and overwrite $k_i$ with a different value.