A blockchain-based privacy-preserving auditable data structure framework

Abstract—**Every digital process needs to consume some data in order to work properly. It is very common for applications to rely on external data sources, such as APIs. When data is not self-generated, the reliability of both the external data source and its produced data cannot be taken for granted. Therefore, ensuring the trustworthiness and verifiability of the received data is paramount. While authenticated data structures are commonly used to establish trust by authenticating the data source and generating proofs of data authenticity or integrity, they fall short in use cases like data notarization that require also verification of data history and its consistency. This problem seems to be unaddressed by current literature, which proposes some approaches aimed at executing audits by internal actors with prior knowledge about the data structures. In this paper, we analyze the terminology and the current state of the art of the auditable data structures, then we propose a general framework that makes use of a public blockchain as trusted anchor for notarizing data, thereby supporting privacy-preserving audits from both internal and external entities without prior data knowledge. A detailed description of the framework implementation, alongside with experimental results, is provided.**

Index Terms—**Auditable data structures, Data consistency, Consistency proof**

I. Introduction

Data is the foundation of today's technology-driven world. Most technological activities, at some point, needs to process or evaluate some kind of data for their advancement. In real-world scenarios, digital processes often exchange data with external entities, such as cloud service providers, which are assumed to be trusted though this assurance is not always guaranteed. This highlights the user's need for a mechanism that ensures the data received and the operations conducted on that data are both trustworthy and verifiable.

There are well-established lines of research focused on data integrity and authentication when the data publisher or the outsourced data manager are untrusted, proving that the data has not been tampered with. On the other hand, the current literature that addresses the auditability of the sequence of operations executed on the data is at an early stage, even if very promising. The lack of standard definitions for key concepts such as data consistency and data history is the main motivation behind this paper. We propose a set of terminology and key concepts aimed at establishing a robust knowledge base for the field of auditable data structures and their application. Additionally, we introduce a framework that generalizes the models currently employed in major existing applications, including commercial products not yet described in scientific literature that make use of auditable data structures to improve the overall trust in the data but also in the sequence of operations leading to it.

The paper is structured as follows: the rest of this section illustrates the motivations behind this work, Section II provides useful information to understand the topic and the proposed framework, Section III reviews a list of related works. Section IV describes our privacy-preserving externally auditable data structure framework in terms of architecture (Section IV-A), a specific implementation (Section IV-B) and experimental evaluations (Section IV-D). Section V concludes the paper.

A. Motivations

Blockchain technology, pioneered by Chaum, Haber and Stornetta [1, 2] and popularized by Bitcoin [3], builds a P2P network whose peers do not know and do not trust each other but work together to maintain a shared state (the ledger of all the transactions between the users). A consensus algorithm defines the rules to upgrade the state, e.g. by deciding which peer will compute and communicate the next chunk of data [4]. The generalization of this technology is also known as Distributed Ledger Technology (DLT), since not all implementations use a chain of blocks (e.g., IOTA utilizes a Direct Acyclic Graph called Tangle [5]). A DLT is public and permissionless when any peer can send and receive the transactions, and join the consensus without restrictions. Private and permissioned DLTs have been developed to address different use cases, shifting the attention to smaller scale system where the peers do know but do not trust each other [6].

Public and permissionless DLTs guarantee high security, decentralization and provide transparent systems; however, they suffer of scalability problems (causing a low transaction throughput), require fees to execute transactions (which can be very expensive in some networks), and do not comply with privacy-preserving regulations. Instead, private and permissioned DLTs typically have higher throughput, lower infrastructural costs, and are able to limit data access. On the other hand, they are less decentralized and the security properties apply only to the internal participants [6]. For industrial use cases the trend is to adopt private and permissioned DLTs [7], with Hyperledger Fabric [8] being one of the most popular frameworks used to build them. In the case of public DLTs, auditability is inherently achieved due to the lack of distinction between internal and external actors, enhanced by the immutability and transparency of such systems [9]. Conversely, in the case of private DLTs there are known issues preventing external auditability: when a piece of data from a private DLT is disclosed for the first time, it is not possible for actors outside of the network to verify its consistency with the contents of the private ledger due to access limitations. More critically, even when full access to the private network is granted to an auditor, it is still possible for the participants of the private network to agree to re-write history right before the auditor would join the network, effectively forking the ledger (a history re-writing fork has been executed publicly on Ethereum in response to the DAO attack [10], a fork can be done more easily, and secretly, on a smaller and controlled network [11]). This problem is present also when a new node joins the private network.

With the framework presented in this paper we aim to address this scenario, in which a new coming node in a network or an external auditor (the crucial point is the absence of a prior data knowledge) wants to verify that the ledger data have a unique history, with no forks and a consistent and verifiable evolution over time.

II. Background

This section is dedicated to explain some preliminary information needed to better understand the content of the paper. We define key terms and concepts, and analyze the two predominant models in both academic literature and commercial applications: the *three-party model*, employed in data replication scenarios, and the more general *two-party model*, utilized in data outsourcing scenarios.

A. Terminology

Here we introduce key terminology and concepts that are crucial for understanding the objectives and the underlying ideas of the privacy-preserving externally auditable data structure framework that we propose in Section IV. Some terms also apply to the context of authenticated data structure models.

- Data structure **operation**: a function defined on the data structure. It can be executed, by taking arguments and returning a result. If the execution can have the side effect to modify the state of the data structure, then it is an **edit operation**, otherwise, it is a **query operation**.
- Data structure **digest**: an identifier of the current state of the data structure. It is usually compact and computed by cryptographic hash composition. In order to be safely used in the context of authenticated or auditable data structures, it must be at least second-preimage resistant (i.e., given a digest derived from a certain data structure state, it should be computationally hard to find a different state identified by the same digest value).
- Data structure **consistency**: the property of two time-ordered data structure instances where the newer one can be produced starting from the older one by executing a sequence of valid edit operations.
- Data structure **history**: a data structure that identifies a chronologically ordered sequence of versions of the same data structure. For instance, a simple data structure history can be a list of digests.
- **History consistency**: the property of data structure history where consistency holds for every pair of data structure versions.
- **Proof**: a statement, paired with some supporting data. The supporting data must be sufficient to independently verify that the statement is true with a high probability, without relying on trust.
- **Integrity proof**: a proof stating that a given operation executed on a data structure with a given digest will produce a given result. Inclusion proofs (e.g., the ones implemented as the hash path from the root to the leaf of a Merkle tree) are a specific case of integrity proofs.
- **Consistency proof**: a proof stating that two time-ordered data structures with given digests are consistent.
- **Audit**: an inspection of a data structure aimed to check some properties of the data structure content in present or past times, and/or some properties of the sequence of edit operations executed on it.
- **External auditor**: an auditor without prior knowledge of the data structure.
- **Externally auditable data structure**: a data structure whose history consistency can be verified by an external auditor.
- **Privacy-preserving externally auditable data structure**: a data structure that is externally auditable, but also preserves data secrecy during audits (i.e., the history must be devoid of data, for example by using one-way data indirections).

B. Three-party model

In [12] the first formal model for authenticated data structures, known as the *three-party model*, was proposed, addressing the data replication scenario.

In the data replication scenario, a data structure is managed by a trusted *source* and replicated by an untrusted *directory* to delegate distribution and also improve scalability. When querying the *directory*, the *user* needs a way to check that responses have not been tampered with. In the three-party model, schematized in Figure 1, the *user* knows a digest of the data structure authenticated by the *source*, and the *directory* produces an integrity proof when responding to queries.

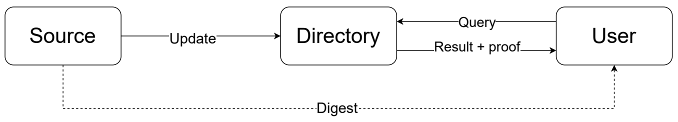

Fig. 1. Scheme of the three-party model for data replication

To be utilized in the three-party model, a data structure needs to meet some key requirements: firstly, it must possess a method for computing a second-preimage resistant digest, which identifies the current state of the data structure; secondly, for each query operation intended for use by the *user*, an algorithm must be defined that generates integrity proofs; finally, a corresponding proof verification algorithm should also be defined.

The three-party model has been adopted by several works as a common framework, useful to describe and compare a large repertoire of authenticated data structures. [13] reviews the major techniques to design authenticated data structures and introduces the three-party mode. In [14], a methodology to use authenticated data structures to authenticate query responses in untrusted scenarios, like the communication between users and directories in the three-party model, is introduced and discussed. Finally, the work proposed in [15] and expanded in [16] proposes to implement an authenticated dictionary as data structure maintained by the directory to support a wider range of possible applications, such as the revocation list management explained in next sections.

C. Two-party model

In [17] the three-party model has been extended to the more general *two-party model*, addressing the data outsourcing scenario, an increasingly common scenario in the context of cloud services.

In the data outsourcing scenario, the *user* delegates the management of a data structure to an untrusted *server* while retaining the ownership. The *user* is not required to maintain a replica of the data structure and needs a way to check that it has not been tampered with. In the two-party model, schematized in Figure 2, the *user* is assumed to know the most recent digest of the data structure; the *server* produces an integrity proof when responding to queries, and a consistency proof when responding to edit operation requests, stating the updated value for the data structure digest.

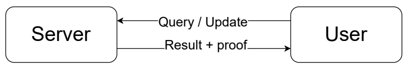

Fig. 2. Scheme of the two-party model for data outsourcing

The key requirements for a data structure to be utilized in the two-party model are essentially the same as those for the three-party model, with an additional one: for every edit operation, an algorithm must be defined that generates consistency proofs, and a corresponding proof verification algorithm should also be defined. This indicates that the requirements for the three-party data replication model are encompassed within those of the two-party data outsourcing model. Hence, the two-party model can be seen as a generalization of the three-party model.

III. Related works

In this section, we review existing applications of auditable data structures, considering both academic literature and commercial solutions. Reviewed applications include digital certificate management, database query verification, log file management, and private distributed ledger auditing. The mentioned applications often implement the three-party data replication model or the two-party data outsourcing model, described in Sections II-B and II-C. The discussion will be focused on how these implementations can handle internal and external auditing and on the data exposed during the audit process. The three approaches most commonly adopted to enable auditing are: edit operations replay, data-exposing proofs, and Merkle proofs. In the edit operations replay approach, an auditor is able to verify any property of the edit operations executed on a data structure in the most straightforward way possible: the full list of edit operations is made available and re-executed on an older version of the data structure already trusted by the auditor, then the identity between the resulting version and the one under audit is checked. Proof-based approaches are more elegant and efficient, but usually it is hard not to expose any data in the process. Merkle proofs appear to be the most suitable proving scheme when addressing use cases with the requirement of data secrecy preservation since they consist only of hashes. Merkle proofs make use of Merkle trees [18] as an underlying data structure that enables the construction of both integrity (inclusion) and consistency proofs. The former is generated in several cases, such as when a log is queried to verify the presence of a certain entry or in authenticated databases in order to prove the presence of a certain record; the latter can be generated, for example, to prove that a newer version of a log is consistent to a trusted older version, meaning that all the entries from the older one are preserved, in the same order, and are a prefix for the entry list of the newer log.

A. Digital certificate management systems

According to [19] digital certificate management system can be modeled as a three-party data replication model where the source is represented by the certification authority (CA) and the directory role is taken by one or more untrusted parties that serve as certificate databases. Laurie and Kasper provided an innovative way to use verified data structures to handle certificate transparency [20] and revocation [21], with the final goal to make the most difficult as possible to a CA to release a certificate for a domain without the domain owner to know about that. Certificates are managed as a log file, where every certificate is appended when released. When required, certificates have to be accompanied with an integrity proof in one or more of these log files. Logs are defined as a Merkle tree data structure containing signed certificates as leaf nodes, and periodically (according to a time parameter) logs produce a digest made by the Merkle root, the number of entries, a timestamp and the log digital signature. CAs are also required to be able to provide integrity proof for each entry every time an auditor issues a query for it. Such integrity proof is a classical Merkle inclusion proof made by the path from the root to the corresponding leaf node.

Regarding the certificate revocation, the authors introduced the deletion operation to the mechanism used with the certification transparency. Logs still have to be able to prove the consistency of the history, meaning that a certificate could be or not be present in a particular version of the log, keeping a valid history for itself between each version. This can be done with a Merkle tree where every leaf node contains pairs from a sorted list of revoked certificates, or by using sparse Merkle trees [22] to store the status on every possible certificate.

B. Log management systems

At a high level, a log file can be modeled as a Merkle tree where every new event captured by the log is a leaf node. To guarantee that the log works properly, the appending of a new element (which equals to a new event captured by the log) is the only valid operation. With such configuration, a log can be used as an externally auditable data structure, and the consistency between any two version of the same log file can always been proved by computing Merkle consistency proofs. Auditability is a key property for a log management system in order to improve transparency and to guarantee the tamper-evidence of the log itself [23]. In [24], three verifiable data structures that can be used to implement a verifiable log management system are proposed:

- **Verifiable log**: a verifiable log is an append-only tree data structure designed to be used, as the name suggests, for log management. This persistent data structure starts empty and generates a new version of itself every time a new row is appended. Periodically, an authenticated digest is published, which includes the tree root hash value, the tree size, and a digital signature. The tree can be queried to check entry inclusion, return all the entries in the tree, and verify the append-only property.
- **Verifiable map**: a verifiable map is a tree data structure where leaf nodes are key-value pairs. Besides the differences between stored data format, verifiable maps work similarly to verified log and are implemented as sparse Merkle tree [22]. Periodically, an authenticated digest is generated from an instance of the map, including the root hash and a digital signature. A verifiable map can be queried to verify the inclusion of a certain key, to retrieve a value by key, and to enumerate all the key-value pairs.
- **Log-backed map**: this data structure is a combination of a verifiable log and a verifiable map: the map is populated with key-value pairs, while the log describes an ordered list of edit operations, which can be replayed in order to verify the history consistency of the map. For this data structure, the digest production has to keep track of the log version related to the specific map version. This hybrid design address the verification of data structure consistency between two different versions of the map by adopting the edit operation replay approach (using the operations included in the log), while the log supports data structure consistency verification by adopting the Merkle proof approach.

Trillian[1] is a commercial log management provider that uses the aforementioned verified data structures [25] to implement a log system, with the goal of guaranteeing log immutability and to provide an auditable event history. Trillian design allows the creation of integrity proofs for query results, and consistency proofs to make it possible to externally verify the data structure consistency between consecutive versions of the log.

[1]https://transparency.dev/#trillian [Accessed on 01 December 2023]

C. Auditable database providers

The main role of auditability for databases is to create a tamper-evidence database, generating inclusion proofs for each record contained in any query response, and also consistency proofs to verify the transaction history. LedgerDB [26] is a centralized database service that offers tamper-evidence, non-repudiation and strong auditability. In LedgerDB, auditability is guaranteed by using a data structure called *journal*, related to the database, and by implementing a transaction execution flow that ensures persistency and tamper-resistance of the processed transactions. When an internal client requests an audit, the journal is populated with client-related data fields (such as client ID, nonce and timestamp) and operator related fields (URI, type, transaction). A hash for the request is calculated according to such data and signed by the client. When preparing a response, the database server adds some fields such as the data stream location and a timestamp. Finally, the server calculates the entire data structure hash, signs it and includes it in the *journal receipt* returned to the client. This protocol ensures that an internal client can always verify the operation validity, using the journal sequence as an operation log. With this setup, LedgerDB supports external audits for transaction inclusion, notary time anchors and verified data removal. By requiring access to the journal and its underlying data, accordingly to the definitions provided in Section II-A, an auditor must be internal to the system. Therefore, LedgerDB does not support external auditability of history consistency of the database. QLDB (*Quantum Ledger Database*) [27] is a document based ledger database developed by Amazon that provides immutable, tamper-proof transaction log useful to track database changes and to generate a unique and complete change history. The database is backed with an append only log, meaning that every change is verifiable and auditable. Users can ask for the document digest, calculated as a SHA-256 hash of the document, and use it to verify the document integrity. It is also possible to verify the document history to check the data history. This setup allows easy internal audits, and also external audits with the drawback to share data with the auditor. Consequently, such external audits are not privacy-preserving.

IV. Privacy-preserving externally auditable data structure framework

The historical roots of the privacy-preserving externally auditable data structure framework that we are introducing can be found in the line of research of the authenticated data structures; they have first been proposed as a solution to safely distribute certificate revocation lists [21]. The use of authenticated data structures, alongside with a verifiable set of operations such as insertion and deletion, makes the entire revocation process more transparent, improving the auditability of the revocation list. This is the principle behind the previous analyzed two-party and three-party models. Our framework aims to further generalize the two-party model, introducing a *trusted storage* that allows the construction of privacy-preserving consistency proofs.

A. Framework overview

In this section, we describe the scenario of privacy-preserving external audit of history consistency, a scenario that extends the two-party authenticated data structure model, for which we propose the privacy-preserving externally auditable data structure framework, schematized in Figure 3.

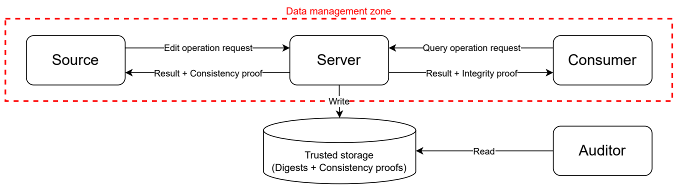

Fig. 3. Scheme of the externally auditable data structure framework

In this scenario, the *source* delegates the management of a data structure to an untrusted *server* while retaining the ownership. The *consumer* query the untrusted *server*. The external *auditor* verifies the history consistency of the data structure managed by the *server*. In the privacy-preserving externally auditable data structure framework, the *server* produces an integrity proof when responding to queries, and a consistency proof when responding to edit operation requests, stating the updated value for the data structure digest. The *server* also writes to the trusted *storage* an authenticated data structure history, including a proof stating its consistency, extending it incrementally whenever an edit operation is executed. The external *auditor* verifies history consistency thanks to the data from the trusted *storage*, including the digest list and the consistency proofs between each pair of consecutive digests. Therefore, there is no need to interact with the *server* when executing an audit. The key requirements for a data structure to be utilized in this framework are essentially the same as those for the two-party model, with the additional one that the consistency proofs must preserve data privacy. Hence, this framework can be seen as a generalization of the two-party data outsourcing model.

B. Implementation

In this section, we provide a detailed analysis about an implementation of the framework described in the previous section. We consider, as our use case, a platform based on the architecture proposed in [28]: a hybrid blockchain architecture where the users of the network generate data structured as blockchain-like append only *ledgers*, and interact with them in a private environment. The ledgers are maintained by the server, and implemented in form of a Merkle tree. For simplicity, here we consider the notarization of a single ledger instance. Periodically, the ledger's Merkle tree digest, along with a Merkle consistency proof between it and the previous value of the digest, is notarized by publishing it on a public blockchain. This approach addresses the auditability issues of private blockchains (discussed in Section I-A) by making this information available on a public blockchain for both internal and external auditors. Given that the sequence of digests, coupled with consistency proofs for consecutive digest pairs, is sufficient to prove the uniqueness of the data history, our system fully supports privacy-preserving external audits. We define the entities explained in the Figure 3 according to a real use case:

- **Source/Consumer:** sources and consumers are the members of the private blockchain network that have permission to execute operations on the ledger. For simplicity, we consider a configuration of the private network where every member have both read and write permissions, removing the distinction between source and consumer.
- **Server:** an intermediary authority with the capability to write on the public blockchain. It is tasked with managing the data structure and associated proofs. Referring to [28], this server entity is embodied by the Notary module.
- **Trusted storage:** we use a public blockchain network as the trusted storage, leveraging the inherent transparency and immutability characteristics of this kind of blockchains. Specifically, Algorand[2] is employed as the trusted storage in our implementation.
- **Auditor:** anyone can be considered an auditor, including internal members of the network or external user with no prior knowledge of the ledger. This latter case could be represented by a new comer member who is interested in joining the private network (the most common case), but in general whoever needs to verify the data history consistency can be considered an external auditor.

In the considered use case, the source and the consumer can interact with the data or execute queries the same way, without any particular constraint except to be part of the network. Data can be edited or queried: edit operations append new data blocks to the ledger, while queries include read operations that does not modify the data. Sources and consumers interact with the server, which returns the corresponding result paired with a proof, generated according to the type of request: consistency proofs for edit operations and integrity proofs for queries.

The server has two main duties in this implementation. First, it periodically executes writes on the public blockchain, to notarize, and then make it transparent and public accessible, the data structure history. A write is executed once every 24 hours and is comprised of three elements: *i)* the digest of the current version of the data structure; *ii)* a consistency proof between the previous published digest and the current one; *iii)* the server digital signature. Digests of the data structure are computed as the cryptographic hash of the root of the Merkle tree. The consistency proof is needed to ensure that the last computed version of the digest is coherent and consistent with the previous one. The digest sequence, paired with the sequence of consistency proofs for each couple of digests, makes it possible to audit the data structure history without requiring a prior knowledge on the content of the structure. The three elements are published on Algorand according to its transaction format, using the *note* field to store the consistency proof; when the consistency proof size is less than 1 kB (the

[2]https://developer.algorand.org/ [Accessed on 01 December 2023]

Algorithm 1: Consistency proof generation

```
Function $createConsistencyProof(tree_1, tree_2)$:
    $leaf \leftarrow$ right-most leaf of $tree_1$;
    $node \leftarrow$ leaf of $tree_2$ that is equals to $leaf$;
    $proof \leftarrow$ empty list;
    while $node$ has a parent do
        $siblingHash \leftarrow node$ sibling;
        if $siblingHash$ is the right-sided sibling then
            $proof.append(siblingHash, \text{“}right\text{”})$;
        else
            $proof.append(siblingHash, \text{“}left\text{”})$;
        $node \leftarrow node$ parent;
    return $proof$
```

Algorithm 2: Consistency proof verification

```
Function $verifyConsistProof(root_1, root_2, proof)$:
    $hash_1 \leftarrow null$;
    $hash_2 \leftarrow null$;
    foreach $item \in proof$ do
        if $item.side ==$ “$left$” then
            $hash_1 \leftarrow Merge(item.hash, hash_1)$;
            $hash_2 \leftarrow Merge(item.hash, hash_2)$;
        else
            $hash_2 \leftarrow Merge(hash_2, item.hash)$;
    return $hash_2 == root_2$ & $hash_1 == root_1$
```

maximum data size of the *note* field), a single transaction is sufficient, otherwise multiple linked transactions are used. The second main duty of the server is to collect source and consumer operation requests, execute them, and generate a corresponding result, which is the digest of the modified data structure for edit operation requests and the query result for query operation requests. Any new digest is paired with a consistency proof, produced by the server, assessing that the new digest is consistent with the previous one. Instead, query results are paired with a data integrity proof.

The trusted storage acts as anchor for data notarization. The introduction of a public blockchain as trusted storage is necessary to make the data history and the consistency proofs immutable and transparent, and therefore to enable privacy-preserving external audits, achieving higher generalization level compared to the two-party data outsourcing model.

C. Proofs management

As stated in the previous section, the server produces a proof every time it responds to a user. Proofs are necessary to authenticate the server in the context of the data management zone. Proofs have several goals in this architecture: first, since the server is not trusted, a mechanism that allows users (sources and consumers equally) to trust the server is needed, and proofs can address this issue. Similarly, when the server publishes a digest on the blockchain, it has to provide also a proof that assess the consistency of the newer digest with the older. In this way, external auditors can rely just on the information published on the blockchain to make an audit with no prior data knowledge, without the need of use information from untrusted sources. The server produces two types of proofs: inclusion proofs and consistency proofs. Inclusion proofs are generated when a consumer queries the server asking for a subset of data contained in the ledger. Since the ledger is maintained as a Merkle tree by the server, this proof is generated as a simple Merkle inclusion proof, that is the hash path from leaf up to the root for each requested data block. On the other hand, consistency proofs are required to assess that the blocks created by edit operations are appended correctly. To prove the consistency of two different version of an append-only data structure, given T_1 and T_2, Merkle roots of two chronologically ordered versions of the same data structure, with T_2 the root of the newer version, we aim to prove that T_2 is the Merkle root of a data structure that will be obtained by only appending new blocks to the data structure identified by T_1, meaning that the ordered sequence of leaves rooted by T_1 is a prefix of the ordered sequence of leaves rooted by T_2. In practice, the consistency proof is an inclusion proof of the rightmost leaf rooted by T_1 in the Merkle root T_2. In particular, the proof is calculated according to the pseudo-code provided in Algorithm 1: the computation starts from the rightmost leaf rooted by T_1, and build the hash path in the tree rooted by T_2: the produced proof contains the hash values of the sibling nodes, walking bottom-up from the starting leaf up to the root T_2. Every hash of the hash path is coupled with the sibling side specification: this is needed to verify the proof, since left siblings play an important role in the proof verification process. The verification, illustrated in Algorithm 2, consists in the computation of two values using data contained in the proof. Two hash aggregations are executed at the same time: the first one aims to reproduce the value T_2 computing a full hash path merge, while the second aims to reproduce T_1, i.e. the older version root value, aggregating only the hash values from left siblings in T_2. This ensures that the ordered sequence of leaves rooted by T_1 is a prefix of the ordered sequence of leaves rooted by T_2.

D. Experimental evaluations

We present an experimental evaluation based on a series of tests conducted on the prototype implemented as described in the preceding sections. These experiments aim to assess the prototype's performance and its applicability in real-world industrial scenarios, in particular the one based on a hybrid blockchain architecture previously described in Section IV-B. Our focus was on the notarization process of a single ledger instance. Key aspects evaluated include the size of data committed on the blockchain relative to the ledger's growth, the time efficiency in generating and verifying proofs, the cost associated with each commit, and the scalability of the audit procedure. Table I details the experimental setup, considering three parameters: n, the initial number of data blocks in the ledger; d, representing the number of blocks added per run; and *runsize*, indicating the iteration count for each set of n and d values. Each individual run is executed by first calculating n_0, which is the initial length of the ledger, randomly selected from a uniform distribution ranging between $n \cdot \frac{2}{3}$ and $n \cdot \frac{4}{3}$. A ledger of n_0 blocks with random content is created, and its digest r_0, the Merkle root value, is computed. Then, d_0 is determined, randomly selected from a uniform distribution ranging between $d \cdot \frac{2}{3}$ and $d \cdot \frac{4}{3}$. d_0 blocks of random content are appended to the ledger and the new digest r_1 is calculated. Finally, a consistency proof between r_0 and r_1 is generated and verified.

TABLE I
PARAMETERS FOR THE EXPERIMENTS

d	n	$runsize$
1	100 – 10.000.000	500.000
10	1.000 – 10.000.000	500.000
100	10.000 – 10.000.000	500.000
1.000	100.000 – 10.000.000	500.000
10.000	1.000.000 – 10.000.000	500.000
100.000	10.000.000	500.000

Through the execution of runs as outlined, we gathered and analyzed various performance metrics for our prototype. For each (n, d) parameter pair, we calculated the average size of the consistency proofs and the number of transactions needed for notarization on the public blockchain, averaged across all runs with identical parameter values. Additionally, we measured the time required to generate the consistency proof, as well as the time needed for its verification. Lastly, we estimated the time necessary to conduct an external audit on a sequence of digests and proofs recorded on the blockchain. All experiments were conducted on a computer equipped with a 12th Gen Intel Core i7-1260P processor, running at 2100 MHz, featuring 12 cores, and equipped with 32 GB of RAM.

1) Proof size: As the first metric, we measured the size of the generated consistency proof. For the hash function, we adopted *SHA-512* [29], ensuring that the hashes and digests are consistently 512 bits in length. Recalling that a consistency proof consists of a list of pairs $(siblingHash, side)$, we have adopted a serialization format in which each elements is 513 bits in size (512 for the hash and 1 for the $side$), with the number of elements not explicitly stated but instead implicitly determined by the length of the file containing it. In line with the theoretical framework for Merkle consistency proofs, the experimental results displayed in Figure 4 demonstrate that the mean size of the proofs is linearly dependent on the sum of the logarithm of the initial number of leaves plus the logarithm of the number of appended leaves. To evaluate the number of Algorand transactions required for notarization on the public blockchain, we recall that our implementation writes them in the $note$ field of the Algorand transaction format, which is fixed at 1 kB in size. The data collected from the experiment show that for many ledger usage scenarios, a single transaction is sufficient to notarize the proof. However, for scenarios with a high frequency of writing, two or more transactions may be necessary. Nevertheless, since the growth of the proof size is essentially logarithmic, the solution can be considered adequately scalable.

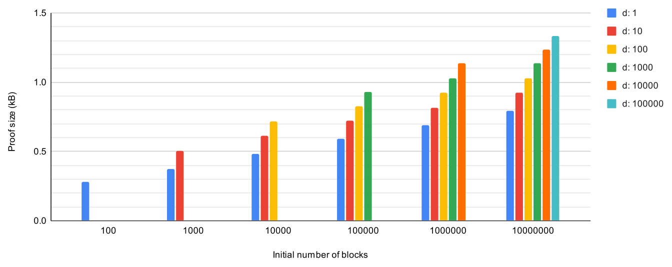

Fig. 4. Variation of proof size with parameters n and d

2) Proof generation time: In our prototype, we adopted the technique of maintaining not only the current digest of each ledger, which is the value of the associated Merkle tree root, but also some additional internal node values of the Merkle tree. Specifically, we retain the values of the nodes that are roots of the highest perfect binary sub-trees into which the tree can be decomposed, which we call the Merkle frontier. These sub-trees correspond bijectively to the bits set to one in the binary representation of the number of leaves, which means we maintain at most $log_2(n)$ of them. The algorithm for updating the Merkle frontier after appending a node is trivial and has a complexity of $O(log(n))$, an operation we perform in the prototype. When generating a consistency proof, we can therefore assume we have available: *i)* the Merkle frontier for the initial version of the ledger; *ii)* the values of the leaves that have been newly appended; *iii)* the roots of all perfect binary sub-trees whose leaves are only those that have been newly appended (this set, as the Merkle frontier, is updated efficiently at the time of appending a node to the ledger). Under these assumptions, the construction of a consistency proof is very efficient, as at most only one of the sibling nodes whose value needs to be retrieved during the execution of Algorithm 1 is required: the last right sibling, the last one before finding only left siblings, which could possibly be the root of a number of new leaves different from a power of two, which we call l. What is done in this case to calculate the missing node value is to compose, using the usual approach of hash merging, a number of at most $log_2(l)$ already available node values. Figure 5 displays the experimental results for the time taken to generate consistency proofs. We observe an increase in time for cases where the value of parameter d is 100 or higher. This increase is due to the fact that only in these cases the presence of the previously described l leaves is statistically significant, requiring several hash calculations that are more computationally intensive than the other steps of the algorithm.

3) Proof verification time: Figure 6 presents the experimental results for the time taken to verify consistency proofs. It shows that the verification time is directly proportional to the length of the proof. Since proof verification requires hash composition steps, it is noteworthy that in our implementation

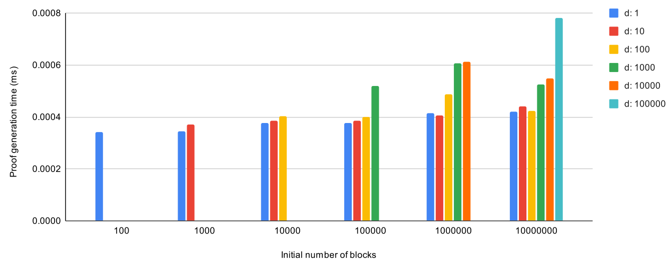

Fig. 5. Variation of proof generation time with parameters n and d

the time required to generate a proof is significantly less than the time required to verify it.

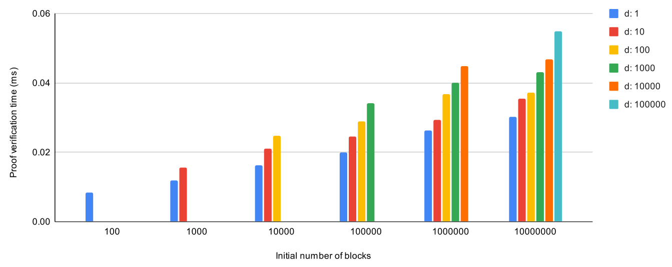

Fig. 6. Variation of proof verification time with parameters n and d

4) External audit time: We estimated the time needed to execute a privacy-preserving external audit of a sequence of digests and proofs on the public blockchain for a ledger of size n. Assuming m notarizations have been performed, meaning m pairs $(digest, proof)$ have been written on the Algorand public blockchain, and each pair is recorded through t transactions, an audit involves the following steps: *1)* retrieve all $m \cdot t$ necessary transactions from Algorand; *2)* verify all m consistency proofs; *3)* ensure each consistency proof starts with the final digest of the preceding proof. The time for step 1 depends on the response time of the Algorand indexer node used to retrieve the transaction list. The total time largely depends on the number of API calls needed: currently, indexer node APIs can fetch up to 1000 transactions in a single call, with pagination for more. For the service offered by PureStake[3], a single API call retrieving 1000 transactions takes about 0.5 seconds. The time for step 2, as experimentally shown, depends on the size of the consistency proofs, which is a function not only of the ledger's size but also of how many blocks are typically added between two consecutive notarizations. The time for step 3 is negligible. Based on experimental data, we observe that for all considered parameter choices of n and d, the average consistency proof verification time is less than 0.001 milliseconds. On the other hand, the average time to retrieve a single consistency proof, considering batch query approach of 1000 items and t being 2, can be estimated at around 1 millisecond. Thus, the verification time of the proofs in our implementation is negligible, and the total time for conducting the audit can be estimated at around 1 second for every 1000 notarization events.

[3]https://www.purestake.com/ [Accessed on 01 December 2023]

V. Conclusion

In this paper, we propose formal definitions and present key concepts pertinent to authenticated and auditable data structures, aiming to establish a solid knowledge base for future research in this promising field. Our discussion begins by analyzing the most important scenarios and models: the three-party model, addressing data replication scenarios, and the two-party model, used in data outsourcing scenarios. Following this analysis, we delve into the state of the art and the current uses of auditable data structures. We focus on a selection of applications, including digital certificates management, log management, auditable database providers, and auditable private DLT systems. Then, we propose a new, more general framework for the privacy-preserving external auditability of data structures. This framework extends the authenticated data structure models by introducing a public blockchain as trusted storage for committing cryptographic proofs from untrusted environments, thereby addressing the need for external audit of history consistency while preserving data secrecy. Finally, we provide an implementation of this framework along with experimental evaluations to demonstrate its effectiveness.

As we are already working on enhancing our implementation to efficiently notarize large collections of ledgers, in the future we plan to provide a detailed documentation for this advanced solution. This extension will involve studying the impact on various aspects such as the data structures involved, the size of notarization data, performance metrics, and the identification of additional relevant real-world applications.

VI. Acknowledgements

Hidden for double blind review

VII. Disclaimer

Hidden for double blind review

References

[1] Alan T. Sherman, Farid Javani, Haibin Zhang, and Enis Golaszewski. "On the Origins and Variations of Blockchain Technologies". In: *IEEE Secur. Priv.* 17.1 (2019), pp. 72–77. DOI: 10.1109/MSEC.2019.2893730. URL: https://doi.org/10.1109/MSEC.2019.2893730.

[2] Stuart Haber and W. Scott Stornetta. "How to Time-Stamp a Digital Document". In: *J. Cryptol.* 3.2 (1991), pp. 99–111. DOI: 10.1007/BF00196791. URL: https://doi.org/10.1007/BF00196791.

[3] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: *Decentralized Business Review* (2008), p. 21260.

[4] Yang Xiao, Ning Zhang, Wenjing Lou, and Y. Thomas Hou. “A Survey of Distributed Consensus Protocols for Blockchain Networks”. In: *IEEE Commun. Surv. Tutorials* 22.2 (2020), pp. 1432–1465. DOI: 10.1109/COMST.2020.2969706.

[5] *The Tangle*. https://wiki.iota.org/learn/about-iota/tangle Last accessed on 22 May 2023. 2022.

[6] Karl Wüst and Arthur Gervais. “Do you Need a Blockchain?” In: *Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018*. IEEE, 2018, pp. 45–54. DOI: 10.1109/CVCBT.2018.00011.

[7] Julien Polge, Jérémy Robert, and Yves Le Traon. “Permissioned blockchain frameworks in the industry: A comparison”. In: *ICT Express* 7.2 (2021), pp. 229–233. DOI: 10.1016/j.icte.2020.09.002.

[8] *A Blockchain Platform for the Enterprise*. https://hyperledger-fabric.readthedocs.io/ Last accessed on 22 May 2023. 2020.

[9] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci. “A blockchain based approach for the definition of auditable Access Control systems”. In: *Comput. Secur.* 84 (2019), pp. 93–119. DOI: 10.1016/j.cose.2019.03.016.

[10] David Siegel. *Understanding The DAO Attack*. https://www.coindesk.com/learn/understanding-the-dao-attack/ Last accessed on 22 May 2023. 2016.

[11] Omar Dib, Kei-Leo Brousmiche, Antoine Durand, Eric Thea, and Elyes Ben Hamida. “Consortium blockchains: Overview, applications and challenges”. In: *International Journal On Advances in Telecommunications* 11.1&2 (2018), pp. 51–64.

[12] Premkumar T. Devanbu, Michael Gertz, Charles U. Martel, and Stuart G. Stubblebine. “Authentic Third-party Data Publication”. In: *Data and Application Security, Development and Directions, IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security, Schoorl, The Netherlands, August 21-23, 2000*. Ed. by Bhavani Thuraisingham, Reind P. van de Riet, Klaus R. Dittrich, and Zahir Tari. Vol. 201. IFIP Conference Proceedings. Kluwer, 2000, pp. 101–112. URL: https://doi.org/10.1007/0-306-47008-X%5C_9.

[13] Roberto Tamassia. “Authenticated Data Structures”. In: *Algorithms - ESA 2003, 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003, Proceedings*. Ed. by Giuseppe Di Battista and Uri Zwick. Vol. 2832. Lecture Notes in Computer Science. Springer, 2003, pp. 2–5. URL: https://doi.org/10.1007/978-3-540-39658-1%5C_2.

[14] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. “Persistent Authenticated Dictionaries and Their Applications”. In: *Information Security, 4th International Conference, ISC 2001, Malaga, Spain, October 1-3, 2001, Proceedings*. Ed. by George I. Davida and Yair Frankel. Vol. 2200. Lecture Notes in Computer Science. Springer, 2001, pp. 379–393. URL: https://doi.org/10.1007/3-540-45439-X%5C_26.

[15] Michael T Goodrich, Roberto Tamassia, and Andrew Schwerin. “Implementation of an authenticated dictionary with skip lists and commutative hashing”. In: *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX’01*. Vol. 2. IEEE. 2001, pp. 68–82.

[16] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. “Improving Authenticated Dynamic Dictionaries, with Applications to Cryptocurrencies”. In: *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*. Ed. by Aggelos Kiayias. Vol. 10322. Lecture Notes in Computer Science. Springer, 2017, pp. 376–392. URL: https://doi.org/10.1007/978-3-319-70972-7%5C_21.

[17] Charalampos Papamanthou and Roberto Tamassia. “Time and Space Efficient Algorithms for Two-Party Authenticated Data Structures”. In: *Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings*. Ed. by Sihan Qing, Hideki Imai, and Guilin Wang. Vol. 4861. Lecture Notes in Computer Science. Springer, 2007, pp. 1–15. URL: https://doi.org/10.1007/978-3-540-77048-0%5C_1.

[18] Haojun Liu, Xinbo Luo, Hongrui Liu, and Xubo Xia. “Merkle Tree: A Fundamental Component of Blockchains”. In: *2021 International Conference on Electronic Information Engineering and Computer Science (EIECS)*. 2021, pp. 556–561. DOI: 10.1109/EIECS53707.2021.9588047.

[19] Moni Naor and Kobbi Nissim. “Certificate revocation and certificate update”. In: *IEEE J. Sel. Areas Commun.* 18.4 (2000), pp. 561–570. DOI: 10.1109/49.839932.

[20] Ben Laurie, Eran Messeri, and Rob Stradling. “Certificate Transparency Version 2.0”. In: *RFC* 9162 (2021), pp. 1–53. DOI: 10.17487/RFC9162.

[21] Ben Laurie and Emilia Kasper. “Revocation transparency”. In: *Google Research, September* 33 (2012).

[22] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. “Efficient Sparse Merkle Trees: Caching Strategies and Secure (Non-)Membership Proofs”. In: *IACR Cryptol. ePrint Arch.* (2016), p. 683. URL: http://eprint.iacr.org/2016/683.

[23] Yanwen Bao, Yongjian Wang, Zhongzhi Luan, Xiang Pei, and Depei Qian. “IndexTree: An Efficient Tamper-Evidence Logging”. In: *12th IEEE International Conference on High Performance Computing and Communications, HPCC 2010, 1-3 September 2010, Melbourne, Australia*. IEEE, 2010, pp. 701–706. DOI: 10.1109/HPCC.2010.41.

[24] Adam Ejidenberg, Ben Laurie, and Al Cutter. *Verifiable Data Structures*. 2015.

[25] *Trillian Verifiable data structures documentation*. https://transparency.dev/verifiable-data-structures/ Last accessed on 22 May 2023.

[26] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. "LedgerDB: A Centralized Ledger Database for Universal Audit and Verification". In: *Proc. VLDB Endow.* 13.12 (2020), pp. 3138–3151. DOI: 10.14778/3415478.3415540. URL: http://www.vldb.org/pvldb/vol13/p3138-yang.pdf.

[27] *Amazon Quantum Ledger Database documentation*. https://docs.aws.amazon.com/qldb/latest/developerguide/ql-reference.docs.html Last accessed on 22 May 2023.

[28] Andrea Canciani, Claudio Felicioli, Andrea Lisi, and Fabio Severino. "Hybrid DLT as a data layer for real-time, data-intensive applications". In: *arXiv preprint arXiv:2304.07165* (2023).

[29] Shay Gueron, Simon Johnson, and Jesse Walker. "SHA-512/256". In: *Eighth International Conference on Information Technology: New Generations, ITNG 2011, Las Vegas, Nevada, USA, 11-13 April 2011*. Ed. by Shahram Latifi. IEEE Computer Society, 2011, pp. 354–358. URL: https://doi.org/10.1109/ITNG.2011.69.