ZipperChain White Paper

Draft Version 0.1

BLOCKY, Inc.

research@blocky.rocks

Abstract

Distributed ledger technologies (DLTs) rely on distributed consensus mechanisms to reach agreement over the order of transactions and to provide immutability and availability of transaction data. Distributed consensus suffers from performance limitations of network communication between particpating nodes. BLOCKY ZipperChain guarantees immutability, agreement, and availability of transaction data, but without relying on distributed consensus. Instead, its construction process transfers trust from widely-used, third-party services onto ZipperChain's correctness guarantees. ZipperChain blocks are built by a pipeline of specialized services deployed on a small number of nodes connected by a fast data center network. As a result, ZipperChain transaction throughput approaches network line speeds and block finality is on the order of 100 ms. Finally, ZipperChain infrastructure creates blocks centrally and so does not need a native token to incentivize a community of verifiers.

1 Introduction

Distributed ledger technologies (DLTs), or blockchains, make possible a growing number of decentralized alternatives to traditionally centralized services. Proof-of-Work blockchains made the development of decentralized services only marginally practical due to high delay, low transaction throughput, and high and unpredictable cost of recording and executing transactions. Newer blockchain proposals address these limitations with novel consensus mechanisms, faster block distribution techniques, and less speculative transaction pricing [19, 25, 38]. Fundamentally, however, the performance of blockchains is limited by their reliance on distributed consensus algorithms, which suffer from performance limitations of network communications between participating nodes [28]. As a result, DLTs rely on widely distributed infrastructure to provide strong security, but at the cost of low transaction throughput, or centralize their infrastructure to improve its network performance and provide higher transaction throughput, but at the cost of weaker security.

Fundamentally, blockchains guarantee immutability, agreement, and availability. Blockchains provide immutability by cryptographically linking blocks in a way that makes their retroactive modification without renewed distributed agreement near-impossible. Agreement, in turn, comes from a blockchain's distributed consensus protocol, which ensures that the creation of new blocks follows preset rules. Finally, availability comes from the replication of blockchain state among distributed nodes that prevents its deletion and, in the case of public blockchains, provides censorship resistance. While these mechanisms may seem inextricably linked in blockchains, we see them as a bundled solution in need of some refactoring lest blockchain benefits always come at the cost of consistency in distributed systems [17].

To escape this conundrum we propose to unbundle blockchain mechanisms by revisiting the underlying trust relationships between a blockchain and its users. Blockchains are considered trustless peer-to-peer systems, because to use them users do not need to trust each other. We point out, however, that other widely used systems are based on strong, but limited trust relationships. For example, users generally trust certificate authorities' assertions of public keys. Similarly, authentication services based on OAuth 2.0 [24] are generally trusted to issue correct authentication tokens. Such trust relationships emerge from a clear self-interest in the correctness of the service by its provider that goes beyond the benefit of circumventing its one particular use case.

In this paper we demonstrate that limited trust in third-party services can give rise to novel mechanisms for immutability, agreement, and availability. We propose a novel blockchain we dub ZipperChain that transfers users' trust in these third-party services onto its correctness guarantees. We build ZipperChain blocks on a pipeline of specialized services deployed on a small number of nodes connected by a fast data center network. The result is a blockchain with strong correctness guarantees based on correctness of third-party service, transaction throughput that approaches the line speed of data center networks, and block finality close to 100 ms. Since ZipperChain infrastructure creates blocks centrally, it does

Symbol	Meaning	Format/Type
a	Enclave attestation	$\langle K, g, ... \rangle$
$\mathcal{A}$	Enclave Service	AWS Nitro Enclave
b	Block	$\langle u, m^H, t^H \rangle$
B	Set of blocks	$\{b_0, b_1, ...\}$
c	Batch number	`uint`
d	Transaction data	`byte[]`
f	Certificate	$\langle b_0, K_{\mathcal{S}}^+, d^H, \bar{t}, \underline{t} \rangle$
g	Signature	`byte[]`
H	Cryptographic hash function	`SHA3_256`
i	Block height	`uint`
I	Container image	`byte[]`
k	Counter value	`uint`
K^+/K^-	Public/private key	`byte[]`/`byte[]`
m	Merkle tree	`byte[]`
M	Set of Merkle trees	$\{m_0, m_1, ...\}$
p	Physical clock reading	`uint`
s	Sequence attestation	$\langle y, k, g \rangle$; $y = u + H(t)$
S	Set of sequence attestations	$\{s_0, s_1, ...\}$
$\mathcal{S}$	Sequencer Service	Sequencer on $\mathcal{A}$
t	Timestamp attestation	$\langle y, p, g \rangle$; $y = H(b)$
T	Set of timestamp attestations	$\{t_0, t_1, ...\}$
$\mathcal{T}$	Timestamp Service	AWS Cognito
u	Unique block ID	`byte[]`
y	Byte array	`byte[]`
$\{y\}_K$	Encryption of y with K	`byte[]`
$D_K(y)$	Decryption of y with K	`byte[]`

Table 1: Paper notation.

not need a native token to incentivize a community of verifiers. ZipperChain's performance and independence from regulated tokens make it well-suited to the needs of existing and future blockchain applications.

The rest of this paper is organized as follows. Section 2 describes the abstractions and implementation of the trusted services, on which ZipperChain relies. Section 3 describes the structure of ZipperChain and how it guarantees immutability, agreement, and availability. Section 4 describes the implementation of ZipperChain. Section 5 offers a discussion of ZipperChain. Section 6 outlines related work on DLT design and performance. Finally, we conclude in Section 7.

2 Trusted Services

ZipperChain makes a departure from distributed blockchain implementations to provide its guarantees of immutability, agreement, and availability based on trusted third-party services. To ground the presentation of ZipperChain in Section 3, in this section we describe the abstractions and implementations of three trusted services, on which ZipperChain relies. To facilitate further discussion we summarize the notation introduced throughout this paper in Table 1.

2.1 Trusted Timestamp Service

The function of the trusted Timestamp Service is to provide accurate and trustworthy physical clock timestamps. A Timestamp Service $\mathcal{T}$ maintains a public/private key pair $K_{\mathcal{T}}^+/K_{\mathcal{T}}^-$ and an accurate physical clock. A Timestamp Service provides the following abstract interface:

$$t := timestamp(y)$$
$$true/false := validate(K, t)$$

The *timestamp* function takes bytes y as input, reads the physical clock value p, and uses these to create a timestamp attestation t. The attestation is a tuple $t = \langle y, p, g \rangle$, with signature $g = \{H(y,p)\}_{K_{\mathcal{T}}^-}$,[1] where H a cryptographic hash function. The *validate* function takes a key K and a timestamp attestation t as input to determine that y and p are correctly signed, or that $H(t.y, t.p) = D_K(t.g)$,[2] where D is a decryption function. When users trust a Timestamp Service and $validate(K_{\mathcal{T}}^+, t)$ returns *true*, they can trust that the Timestamp Service $\mathcal{T}$ has witnessed bytes $t.y$ at time $t.p$. Note that while the *timestamp* function must execute on the trusted Timestamp Service, the *validate* function may be run by a user as long as $K_{\mathcal{T}}^+$ is well-known.

We implement the Timestamp Service based on a user authentication (auth) service, specifically AWS Cognito [4]. An auth service accepts user credentials (`username` and `password`) and, if these match, produces a signed JSON Web Token (JWT). The JWT contains the `username` and a timestamp, among other fields, signed by the auth service.

To get the auth service to provide timestamps over arbitrary data, we first create a user with `username` $= y$`@bky.sh`, where y represents the input bytes as above. The resulting JWT represents the tuple t. When the auth service is AWS Cognito a user that trusts Cognito trusts that the exact bytes y were seen by AWS at a specific time.

2.2 Trusted Sequencer Service

The function of the trusted Sequencer Service is to provide consecutive sequence numbers to distinct events. A Sequencer Service $\mathcal{S}$ maintains a public/private key pair $K_{\mathcal{S}}^+/K_{\mathcal{S}}^-$ and a unitary, monotonic, arithmetic progression counter. A Sequencer Service provides the following abstract interface:

$$s := sequence(y)$$
$$true/false := check(K, s)$$

The *sequence* function takes as input bytes y representing an unique event ID, increments the counter by one, records the counter value as k, and produces a sequence at-

[1] We use the Kerberos security protocol notation [18, 34].
[2] We use the "." operator for member selection.

testation $s = \langle y,k,g \rangle$, where $g = \{H(y,k)\}_{K_S^-}$.[3] The *check* function takes a key K and a sequence attestation s as input to determine that y and k are correctly signed, or that $H(s.y,s.k) = D_K(s.g)$. When users trust a Sequencer Service and $check(K_S^+, g)$ returns *true*, they can trust that a unique event represented by $s.y$ was witnessed by the Sequencer Service S as the $s.k^{\text{th}}$ event. Similarly to *timestamp*, the *check* function may be executed by a user with a well-known K_S^+.

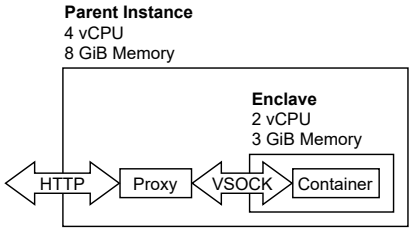


Figure 1: AWS Nitro Enclave.

We implement a Sequencer Service based on the security and correctness guarantees provided by the AWS Nitro Enclaves trusted execution environment (TEE) [5]. Nitro Enclaves create an isolated execution environment inside an Elastic Cloud Compute (EC2) instance based on the same Nitro Hypervisor technology that provides isolation between EC2 instances themselves [11]. Inside an EC2 parent enclave, as shown in Figure 1, an enclave runs a container on its own kernel, memory, and vCPU resources sequestered from the parent instance. An enclave has no persistent storage, interactive access, or external networking. The only means for the parent instance to interact with an enclave is through a VSOCK socket. The parent instance, however, may proxy external requests for example by running a Hypertext Transfer Protocol (HTTP) server. Finally, applications running inside the enclave may request attestations from the Nitro Secure Module (NSM) [9]. An attestation a includes information about the enclave environment recorded as hashes of the continuous measurements of the parent instance ID, the enclave image file (container) as $a.I^H$, and the application requesting the attestation. Optionally, the attestation may also include the public key of the application as $a.K$ and up to 1024 B of user data [7]. NSM packages the attestation as a Concise Binary Object Representation (CBOR)-encoded, CBOR Object Signing and Encryption (COSE)-signed object by the AWS Nitro Attestation key pair K_A^+/K_A^-, where K_A^+ is in a well-known root certificate [6].

We implement a trusted Sequencer Service on a Nitro Enclave as follows. The parent instance runs an API gateway, which exposes the Sequencer Service *sequence* function to clients and proxies calls to it to the enclave S. A *sequence* request includes bytes y representing a unique event ID, which serves as an idempotency key inside S. Upon receiving a *sequence* request the enclave increments an in-memory counter and produces a sequence attestation $s = \langle y,k,g \rangle$ as defined above. It is important to note that repeated requests to sequence the same y will not increment the counter and produce a new sequence attestation; instead the Sequencer Service will serve a previously computed $\langle y,k,g \rangle$.

To demonstrate to clients that the sequence attestation s comes from a Sequencer Service S running inside a Nitro Enclave, the Sequencer Service requests an enclave attestation over its public key K_S^+. The NSM produces an enclave attestation a, whose public key field $a.K = K_S^+$. We make the enclave attestation a well-known.

Upon receiving the sequence attestation s, a client can check it locally. First, the client verifies the authenticity of the well-known enclave attestation a against the well-known key K_A^+. Second, the client calls $check(a.K,s)$ locally and when it returns *true* the client can trust that event $s.y$ was assigned the sequence number $s.k$ by a Sequencer Service S.

It is important to note that in our Sequencer Service implementation the enclave generates the key pair K_S^+/K_S^- on startup, which is distinct across all enclave instantiations. Consequently, every sequence attestation produced by an enclave is unique since the enclave uses a distinct K_S^- to sign an incremented k. As a result it is not possible, even if a Sequencer Service is restarted, to produce two sequence attestations with the same k for different y signed by K_S^-.

Finally, the last issue is that of user trust. A user may trust that the AWS Nitro Enclaves system works correctly. The Sequencer Service, however, is based on our implementation. We make it possible for anyone to inspect our Sequencer Service implementation at `github.com/blocky/sequencer`. The implementation's code and build tools are publicly available and so a user may perform an audit of the code, build an image I, and take the hash of the image $H(I)$. Recall that the enclave attestation a contains a sighed hash of the image running on the enclave as $a.I^H$. Therefore when a user trusts an implementations of Sequencer Service, the user can verify that a sequence attestation was generated using the trusted implementation by checking that $H(I) = a.I^H$. That is, the hash of the image that the user built matches the hash of the image in the enclave attestation.

2.3 Trusted Replication Service

The function of the trusted Replication Service is to provide stable storage to data objects. Since storage nodes have non-zero mean time between failures (MTBF), storage stability is probabilistic and comes from replication of data objects among nodes with mostly independent failures. A Replication Service provides the following abstract interface:

$$replicate(y)$$
$$y \coloneqq fetch(H(y))$$

[3] Note that we reuse some symbols such as k, when their meaning is clear from member selection that differentiates, for example, the physical timestamp $t.k$ from the sequence number $s.k$.

The *replicate* function takes as input object bytes y and replicates them across storage nodes under $H(y)$ as the retrieval key. The *fetch* function takes the hash of the object bytes $H(y)$ as input and returns the object bytes y from one or more replicas.

Fundamental models of distributed systems commonly assume that nodes have access to stable storage to imply that protocol data survives node failures. To argue the correctness of ZipperChain, we need to both strengthen the notion of stability and make it more specific. We define a *stable storage service* as providing durability, immutability, and verifiability. Durability means that a stored object will remain eventually accessible. Immutability means that a stored object will not change in storage. Verifiability means that a third party may verify that a storage service provides durability and immutability.

We implement the Replication Service based on the correctness guarantees of Write Once, Read Many (WORM) cloud storage systems. WORM systems enable their clients to protect stored data against inadvertent modification, or deletion, to meet regulatory requirements on data retention [1]. Specifically, the AWS Simple Storage Service (S3), Microsoft Azure Blob Storage, and Google Cloud Storage guarantee object immutability through legal/compliance holds on data objects that prevent anyone, including the bucket owner, from deleting or modifying objects [13, 23, 31]. The durability of a storage system is often measured as follows. For example, Amazon, Microsoft, and Google design these services to provide 11 nines of durability by replicating data across availability zones within a region [13, 22, 32]. The 11 nines of durability is a guarantee that any stored item will remain accessible over a year with the probability greater than $1-10^{-9}$.

Finally, cloud storage providers allow, with some custom configuration, to make bucket settings publicly readable, which allows anyone to verify that object holds are enabled. Alternatively, an AWS Nitro Enclave with read-only credentials, may inspect bucket settings and emit publicly verifiable attestations over the bucket's setting.

Although the 11-nines durability guarantee is the industry standard, it is not sufficient by itself for storage of ZipperChain objects. As we detail in Section 4, ZipperChain stores four objects per block. We intend to push ZipperChain performance to 10 blocks per second. If we assume that we want a ZipperChain blockchain to operate for a 100 years, it would generate 1.26×10^{11} objects. For a ZipperChain blockchain to remain viable all these objects must remain accessible. Backblaze presents a details of how cloud storage providers calculate durability [36]. We extend their analysis to calculate the joint durability of a set of objects, or the probability that all the objects retain durability over a period of time [12]. We calculate the durability of ZipperChain objects stored on a single cloud bucket over a 100 years to 4.06×10^{-28} – a woefully insufficient number.

To increase the probability that a ZipperChain chain remains viable, we apply erasure coding to ZipperChain objects to distribute their storage. Using random linear network coding (RLNC) [26] we encode each object into 6 shares and need any 3 to decode the object. We partition these shares across buckets located in six regions, two each for AWS S3, Azure Blob Storage, and Google Cloud Storage, which allows us to assume independence of bucket failures. We use the binomial cumulative distribution function to calculate the durability of a ZipperChain object to 35-nines and the cumulative durability of a ZipperChain chain over a 100 years to 14-nines [15].

2.4 Reliability

In addition to being trusted the design of ZipperChain also requires the Timestamp Service, Sequencer Service, and Replication Service to be reliable in that they can recover from crash failures. While reliability at the cost of temporary unavailability may be assumed for AWS Cognito and cloud storage services, the same cannot be done for an AWS Nitro Enclave. An enclave may crash, but because it relies on an internally generated key pair and in-memory state to provide unique sequence attestations, the enclave may not be restarted. Non-trivial software is inevitably subject to bugs, so it may also be necessary to switch a ZipperChain onto another Sequencer Service node running an update version of its software. To address these issues, we have designed a mechanism for ZipperChain to reliably replace Sequencer Service nodes. We make the description of this mechanism a focus of a separate paper. For the remainder of this paper, however, we assume that the Sequencer Service is reliable, at the cost of no new ZipperChain blocks being possible in the case of a Sequencer Service failure.

3 ZipperChain

We propose a new blockchain we dub ZipperChain that provides immutability, agreement, and availability based on strong, but limited user trust in the Timestamp, Sequence, and Replication services. The key innovation of ZipperChain is its structure and construction process that transfer user trust in these services onto ZipperChain's correctness guarantees.

Figure 2 shows the structure of a ZipperChain blockchain. A block includes the hash of a Merkle tree created from a set of leaves containing transaction data. For example, block b includes the hash of the Merkle tree m as $b.m^H = H(m)$.[4] The height of b is the number of blocks on the path from b to b_0. For convenience, a block b at height i can be referred to as b_i and its height denoted as $b.i$. The blocks also include a universally unique identifier (UUID) [30] field u assigned during block creation.

[4] We use the function "H" to refer to both the process of Merkle tree construction and the process of computing a single hash, such as `SHA3`. It should be clear from the context as to which functionality is intended.

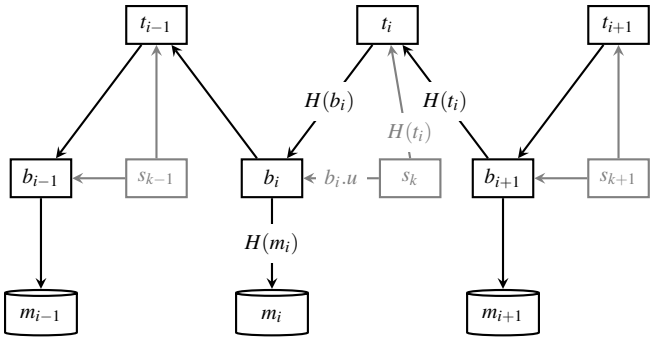


Figure 2: A ZipperChain blockchain formed through links between blocks (b), Merkle trees (m), timestamps attestations (t), and enclave (sequence) attestations (a).

The physical timestamp for each block comes from a timestamp attestation created by a trusted Timestamp Service as described in Section 2.1. For example, the timestamp attestation t_i includes the hash of the block b_i as $t_i.y = H(b_i)$. To incorporate each timestamp attestation into a ZipperChain blockchain, the next block b_{i+1} includes the hash of t_i as $b_{i+1}.t^H = H(t_i)$.

The sequence number for each block comes from its sequence attestation created by a trusted Sequencer Service as described in Section 2.2. For example, the k^{th} sequence attestation s_k includes $b_i.u$, or the unique id of the block b_i. To make dealing with forks possible during the verification process described in Section 4.3 s_k attests both the block and its timestamp attestations as $s_k.y = b_i.u + H(t_i)$.[5]

Given a block b, timestamp attestation t, and sequence attestation s, we call 3-tuple $\langle b,t,s\rangle$ a *triad*. Given a triad $\Delta = \langle b,t,s\rangle$, timestamp service public key $K^+_{\mathcal{T}}$, and sequencer service public key $K^+_{\mathcal{T}}$, we call Δ a *true triad for keys $K^+_{\mathcal{T}}$ and $K^+_{\mathcal{S}}$* if the following conditions conjunctively hold:

1. $\Delta.t.y = H(\Delta.b)$
2. $\Delta.s.y = \Delta.b.u + H(\Delta.t)$
3. $validate(K^+_{\mathcal{T}}, \Delta.t) = true$
4. $check(K^+_{\mathcal{S}}, \Delta.s) = true$

When clear from context, we use the term *true triad* (and omit referencing the keys for which the triad is true). For example, assuming that *validate* and *check* are true for all timestamp and sequence attestations in Figure 2, the triad $\langle b_i, t_i, s_k\rangle$ is a true triad. On the other hand, the triad $\langle b_i, t_{i+1}, s_k\rangle$ is not a true triad because it fails condition 1.

Note that it is possible for multiple ZipperChain chains to coexist. We identify a ZipperChain chain by the unique UUID of its block zero $b_0.u$. Each ZipperChain is associated with a specific Sequencer Service $\mathcal{S}$ identified by its public key $K^+_{\mathcal{S}}$ generated during the startup of $\mathcal{S}$. Just as users can trust the correctness guarantees of one chain, so can multiple chains trust each others' correctness guarantees. However, the total order of transactions is maintained within a chain, but not across chains.

The structure of a ZipperChain blockchain guarantees immutability, agreement, and availability. The sections below outline the intuition behind these claims, while we will eventually provide formal proofs in the Appendix.

3.1 Immutability

ZipperChain guarantees immutability by the alternating structure of blocks and timestamp attestations that come together like the teeth of zipper as a ZipperChain chain grows. A timestamp attestation provides a third-party trusted signature over the hash of the block, which includes the Merkle root constructed from a set of transaction data, also referred to as transactions. Any change to these transactions would be detectable as a mismatch between the block hash and the bytes signed by the timestamp attestation. Thus, as long a timestamp attestation remains a part of a chain, its block remains immutable.

A timestamp attestation remains in the chain, because the following block includes its hash. Thus an attestation at the at end of a chain signs its block and, transitively, the previous attestation and, indirectly, its block, and so on.

A user that considers a block as belonging a ZipperChain blockchain can be sure of the block's integrity by verifying its timestamp attestation. The user can also check the integrity of the chain by verifying the preceding blocks and attestations all the way to a well-known block zero for a particular ZipperChain instance.

3.2 Agreement

ZipperChain guarantees agreement by detecting and eliminating forks, so that all users see the same totally ordered set of block and, by extension, transactions. We define a fork as the existence of two blocks b_i and b'_i with the same block height $i > 0$ and the same previous block b_{i-1}. Figure 3 illustrates a fork in ZipperChain, where blocks b_i and b'_i point to the same previous timestamp attestation t_{i-1} and transitively block b_{i-1}.

Forks, in general, are problematic in blockchains because they create the possibility of an inconsistency of application state represented on the blockchain. Let us assume that two users submit transaction data d and d' that are incompatible with each other. If there is no fork in the chain, any client reading the blockchain sees the same consistent history in which, say, d precedes d' in the same block, or across different blocks. Then according to the rules of an application, the transaction d may be applied to the state and d' may be ignored. If, on the other hand, d and d' are in the forked blocks

[5] We use the "+" operator for concatenation.

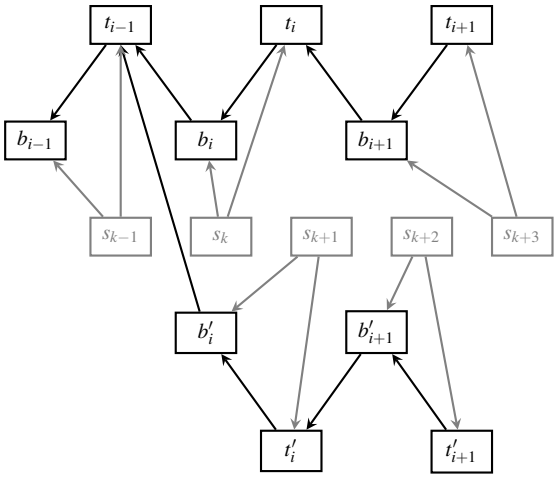


Figure 3: A fork in a ZipperChain blockchain.

b_i and b'_i it is not clear how to order d and d' to determine which transaction to apply and which to ignore.

Forks also create another problem unique to ZipperChain. A user verifying the integrity of a ZipperChain blockchain by walking through it backwards from a given block, may not know whether the encountered blocks are on the main chain, or on a fork. This uncertainty is problematic, because transactions in blocks on a fork will not be considered valid by the users following the main chain. Additionally, the forked blocks may be maliciously deleted (assuming a catastrophic failure of Replication Service) without users on the main chain detecting that deletion.

ZipperChain detects forks by using block heights and sequence attestations to create a total order of blocks. Given two tuples $\langle b,t,a\rangle$ and $\langle b',t',a'\rangle$ of corresponding blocks, timestamp, and enclave sequence attestations such that $s.y = b.u \mathbin{+\!\!+} H(t)$ and $s'.y = b'.u \mathbin{+\!\!+} H(t')$, we define an order relation '$\rightarrow$' as

$$\langle b,s\rangle \rightarrow \langle b',s'\rangle \iff b.i < b'.i \vee (b.i = b'.i \wedge s.k < s'.k).$$

Following this definition the total order of the pairs of blocks and enclave attestations in Figure 3 is

$$\langle b_{i-1},s_{k-1}\rangle \rightarrow \langle b_i,s_k\rangle \rightarrow \langle b'_i,s_{k+1}\rangle \rightarrow \langle b'_{i+1},s_{k+2}\rangle \rightarrow \langle b_{i+1},s_{k+3}\rangle.$$

When there is a fork, we determine the block on the *main chain* using two rules. First, a block may be on the main chain only if its parent is on the main chain. Second, if multiple blocks have a parent on the main chain, the lowest order block is on the main chain. For example, in Figure 3, assume that b_{i-1} is on the main chain and a fork stats at height i. Given $\langle b_i,s_k\rangle$ and $\langle b'_i,s_{k+1}\rangle$ at the start of the fork. We determine which block is on the main chain by observing that $\langle b_i,s_k\rangle \rightarrow \langle b'_i,s_{k+1}\rangle$, which implies (deterministically) that b_i is on the main chain, which b'_i is not. Similarly, given that b_i is on the main chain, users can deterministically decide that b_{i+1} is on the main chain even though $\langle b'_{i+1},s_{k+2}\rangle \rightarrow \langle b_{i+1},s_{k+3}\rangle$, because b_{i+1}'s predecessor is on the main chain, while the predecessor of b'_{i+1} is not.

3.3 Availability

The present invention provides strong, probabilistic guarantees on the availability of data by using a trusted Replication Service to increase the distribution of blocks, timestamp attestations, sequencer attestations, and the leaves of the Merkle trees. Data replication with a trusted Replication Service creates redundant shards of data distributed among independently failing replicas. As a consequence, the encoded data remains available to users even if some of the replicas become unavailable. Even in the case when a sufficient number of replicas is not momentarily available, users remain confident that the stored data remains intact, because of the high durability guarantees of a trusted Replication Service, and will become available again. It is important to note that the present invention records transaction data in a manner that allows users to verify immutability and agreement over blockchain transactions as long as the guarantees of a trusted Replication Service remain in place, even if the other trusted services, or the blockchain creation mechanism, become unavailable.

4 Implementation

ZipperChain provides the following abstract interface:

$$write(d)$$
$$f := verify(b_0, K^+_{\mathcal{S}}, K^+_{\mathcal{T}}, d^H)$$

The *write* function takes the transaction data d as input and starts the process of recording d on a ZipperChain blockchain. The *verify* function takes a ZipperChain's block zero b_0, the public key of the Sequencer Service $K^+_{\mathcal{S}}$, the public key of the Timestamp Service $K^+_{\mathcal{T}}$, and the hash of a transaction $d^H = H(d)$ as input to return a certificate f, which confirms that the transaction exists on a finalized block in the ZipperChain chain $b_0.u$. The certificate contains the transaction hash d^H, two timestamps $\bar{t}$ and $\underline{t}$ representing the upper and lower time bounds for transaction acceptance, and the chain id fields $b_0.u$. When users trust the *verify* function they know that the transaction data d was written onto a ZipperChain chain between $\bar{t}$ and $\underline{t}$ and will remain unchanged on that chain.

Let f and f' be certificates of transaction data d and d', respectively, such that b is the block with hash $f.\underline{t}.b^H$ and b' is the block with hash $f'.\underline{t}.b^H$. Transaction d *precedes* d' if and only if $b.i < b'.i$; or $b.i = b'.i$ and d is to the left of d' in the leaves of the Merkle tree m, such that $H(m) = b.m^H$.

At the moment ZipperChain does not implement a smart contract functionality. However, since ZipperChain provides

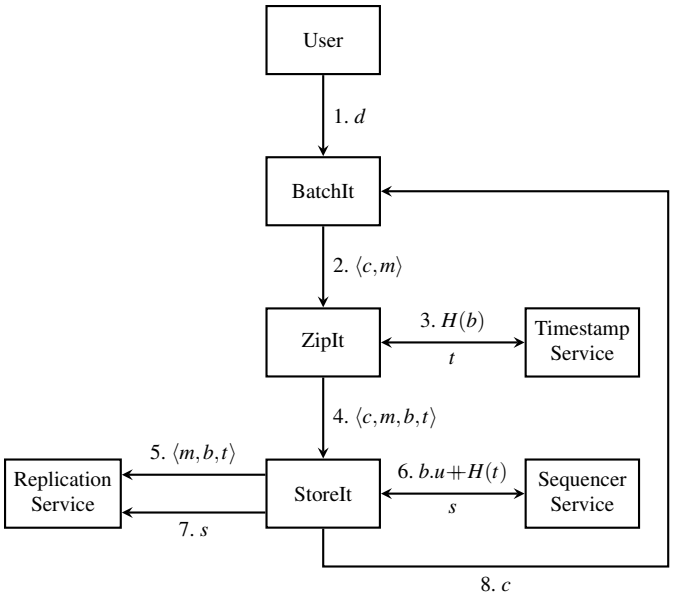


Figure 4: Process implementing the *write* function.

a total order of transactions, if a transaction were the publication, or an invocation of a smart contract, that transaction could be executed deterministically off-chain based on the resulting state of the preceding transactions. Offchain transaction execution has already been successfully demonstrated by several blockchain implementations [20, 25, 40]. We plan to follow a similar path to extend ZipperChain with smart contract functionality. Differently from these blockchains, however, we do not plan on creating a native ZipperChain token. Instead, we plan to support cryptocurrency operations based on coins transferred to a ZipperChain smart contract using a cross-chain approach [29, 37].

4.1 Recording a transaction

Figure 4 shows the process implementing the *write* function in ZipperChain. The numbering of the process description below corresponds to the arrow numbering in the figure.

1. To sign a transaction d, a User calls *write*(d). The BatchIt service receives d and enqueues the *write* request internally.
2. Periodically, the ZipIt service asks BatchIt for a batch of *write* requests. BatchIt dequeues a set of requests and creates a batch identified by a batch ID c. BatchIt then creates a Merkle tree m, where each leaf is a transaction d from a *write* request in the batch. BatchIt then replies to ZipIt with the tuple $\langle c,m\rangle$.
3. Upon receiving a batch from BatchIt, the ZipIt service creates a block $b = \langle u, m^H, t^H\rangle$, where u is a freshly generated UUID, $m^H = H(m)$ is the root of the Merkle tree, and $t^H = H(t_{i-1})$ is the hash of the preceding timestamp attestation. The starting point of a ZipperChain blockchain is a well-known block zero b_0 and its corresponding timestamp attestation t_0, and so ZipIt always has a t_{i-1} to include in a block.
   Next, ZipIt invokes the *timestamp* function of a trusted Timestamp Service by passing it the hash of the block $H(b)$. The attestation service returns a timestamp attestation $t_i = \langle y,p,g\rangle$, where $y = H(b)$ and $g = \{H(y,p)\}_{K_{\mathcal{T}}^-}$. Finally, ZipIt saves t internally as t_{i-1} to include its hash in the next block.
4. ZipIt sends the tuple $\langle c,m,b,t\rangle$ to the StoreIt service.
5. StoreIt sends the tuple $\langle m,b,t\rangle$ for replication to the Replication Service. Crucially, StoreIt does not move onto the next step until the Replication Service confirms that the tuple has been successfully replicated.
6. Once a block has been replicated it is safe to assign it a sequence number. StoreIt send the unique ID of the block $b.u$ concatenated with the hash of the timestamp attestation $H(t)$ to the Sequencer Service, which replies a sequence attestation $s = \langle b.u \mathbin{+\!\!+} H(t), k, g\rangle$, where $g = \{H(b.u \mathbin{+\!\!+} H(t), k)\}_{K_{\mathcal{S}}^-}$.
7. The StoreIt service passes the sequence attestation s to the Replication Service for replication. Upon completing the replication of the enclave attestation, all transactions in the block are finalized.
8. Finally, StoreIt sends the batch id c to BatchIt to notify it that a block corresponding to a batch of requests has been replicated, which allows BatchIt to delete the batch.

It is important to note that the *write* function does not guarantee that a transaction has been recorded on a ZipperChain blockchain. Indeed the *write* function returns after step 1. notifying the User that the transaction has been accepted for processing. The User knows that a transaction has been recorded on the blockchain only after the *verify* function returns a certificate. The User can expect that *verify* will produce the correct result on a transaction only after the transaction is finalized.

4.2 On Main Chain

The On Main Chain algorithm (*omc*) identifies the sequence of triads on the main chain. Let B, T, S be a set of blocks, timestamp attestations, and sequence attestations, respectively. Recall that b_0 is the genesis block, $K_{\mathcal{S}}^+$ is the public key of the sequencer service, $K_{\mathcal{T}}^+$ is the public key of the timestamp service, and a is the well known enclave attestation of the sequencer. The algorithm, $omc(b_0, K_{\mathcal{S}}^+, K_{\mathcal{T}}^+, a, B, T, S)$ produces a set of triads. We provide the details of *omc* in Algorithm 2.

From a high level, Lines 1–10 initialize, verify keys, check the sequence attestation for b_0 and validate the timestamp attestation for b_0. Upon completion of these lines, the main chain Z is initialized with a true triad containing b_0.

Lines 12–26 grows Z using the definition of blocks on main chain. Specifically, to find the next true triad, it is sufficient to find the lowest order block b^* whose parent is the last block

of Z (along with a few additional conditions and supporting objects). Lines 13–18, find a candidate lowest order successor in B, called $\underline{\Delta}.b$, though it may not be b^*. The two may not be the same when the verification process lacks some object data, for example due to replication delay, or download failure. Lines 20–25, use properties of the sequence attestations to rule out the existence of a $b^* \neq \underline{\Delta}.b$.

4.3 Verifying a transaction

After calling the *write* function on a transaction, the User needs to verify that the transaction was written onto a ZipperChain blockchain, i.e. that the transaction exists in a finalized block on the main chain.

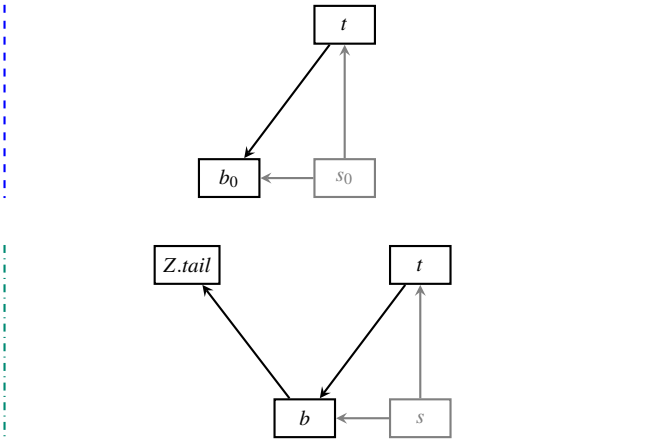
The verification process proceeds as follows:

1. To verify a transaction a User calls $verify(b_0, K_S^+, K_T^+, H(d))$ by passing in transaction data d and the information identifying a ZipperChain blockchain, namely a block zero b_0 and the public key of a Sequencer Service K_S^+, along with the public key K_T^+ of the Timestamp Service. As mentioned earlier, the *verify* function can execute on the User's machine, and so the User does not need to trust BLOCKY to execute the function correctly.
2. The *verify* function downloads the current state of a ZipperChain blockchain from the Replication Service. Specifically, this state comprises the set of blocks (B), Merkle trees (M), timestamp attestations (T), and sequence attestations (S). In practice, these sets and their verification as described below may be cached (bootstrapped) and extended as a ZipperChain chain grows.
3. Next, the *verify* function determines which blocks and timestamp attestations are on the main chain. To do so, *verify* calls the function *omc*, shown in Algorithm 2, which produces a set of alternating main chain blocks and timestamp attestations $Z = omc(b_0, K_S^+, K_T^+, a, B, T, S)$. Recall that a is the enclave attestation over the public key of the sequencer service signed with the well-known public key K_A^+.
4. Finally, the *verify* function determines whether any main chain block contains the transaction d. We assume that transactions are idempotent and their hashes unique, and so the certificate of a transaction always pertains to its first instance. To create a certificate, *verify* calls the *makeCertificate* function, shown in Algorithm 3, which produces a certificate $f = makeCertificate(H(d), M, Z)$. If $f \neq \varnothing$, *verify* returns the certificate to the User.

When the *verify* function returns a certificate f, the function asserts that the transaction d existed before the time $f.\bar{t}$ and after time $f.\underline{t}$ on the ZipperChain ledger $b_0.u$. When *verify* returns a certificate it also indicates that the transaction is final and will not change on the ZipperChain ledger.

Algorithm 2 $omc(b_0, K_S^+, K_T^+, a, B, T, S)$

1: $Z \leftarrow []$ ▷ Main chain triads
2: $k \leftarrow -1$ ▷ Boundary of seen true triad sequence numbers
3: **if** $D_{K_A^+}(a).K \neq K_S^+$ **then**
4: **return** $\varnothing$ ▷ Unattested sequencer service public key
5: ▷ Check block b_0 ◁
6: **if** $\exists b \in B, t \in T, s \in S \mid H(b) = H(b_0) \wedge t.y = H(b) \wedge validate(K_T^+, t) \wedge s.k = 0 \wedge s.y = b.u \# H(t) \wedge check(K_S^+, s)$ **then**
7: $Z \leftarrow Z \# (b, t, s)$ ▷ Add 0th triad to main chain
8: $k \leftarrow 0$ ▷ Mark true triad with seq. num. 0 as seen
9: **else**
10: **return** $\varnothing$ ▷ Block b_0 was invalid
11: ▷ Add main chain triads to Z ◁
12: **loop**
13: ▷ Find all true triad successors to last triad in Z ◁
14: $C \leftarrow \{\langle b, t, s\rangle \mid b \in B \wedge t \in T \wedge s \in S \wedge b.t^H = H(Z.tail.t) \wedge t.y = H(b) \wedge validate(K_T^+, t) \wedge s.y = b.u \# H(t) \wedge check(K_S^+, s)\}$
15: **if** $|C| = 0$ **then**
16: **return** Z ▷ No more successors
17: ▷ Find successor with minimal sequence attestation ◁
18: $\underline{\Delta} \leftarrow \text{argmin}_{c \in C}\, c.s.k$
19: ▷ Fill in the sequence gap ◁
20: **if** $k < \underline{\Delta}.s.k$ **then**
21: ▷ Find true triads between k and $\underline{\Delta}.s.k$ ◁
22: $G \leftarrow \{\langle b, t, s\rangle \mid b \in B \wedge t \in T \wedge s \in S \wedge t.y = H(b) \wedge validate(K_T^+, t) \wedge s.y = b.u \# H(t) \wedge check(K_S^+, s) \wedge s.k \in [k, \underline{\Delta}.s.k]\}$
23: **if** $|G| \neq \underline{\Delta}.s.k - k - 1$ **then**
24: **return** Z ▷ Missing a triad
25: $k \leftarrow \underline{\Delta}.s.k$ ▷ Mark true triads up to $\underline{\Delta}$ as seen
26: $Z \leftarrow Z \# \underline{\Delta}$ ▷ Extend the main chain



Algorithm 3 *makeCertificate*(d^H, M, Z)

```
▷ Find the first block whose Merkle tree contains txn d ◁
for x ← 0, x < |Z|, x ← x + 1 do
    if ∃m ∈ M | d^H ∈ m ∧ Z[x].b.m^H = H(m) then
        ▷ Found d in a block. Return a certificate. ◁
        return ⟨Z[0].b.u, d^H, Z[x-1].t, Z[x+1].t⟩
return ∅ ▷ No main chain block contains transaction d
```

5 Discussion

We point out that ZipperChain, by design, does not have a native token. The continued operation of a ZipperChain instance depends on payments to cloud service providers to operate the ZipperChain construction process and the third-party services on which it relies. As such, ZipperChain does not suffer from regulatory challenges faced by coin-emitting chains nor does it ask users to pay gas fees for blockchain operations. At the same time, a smart contract system supported by ZipperChain could interface with a diverse set of payment rails, including established crypto or fiat currencies, as required by the applications it supports. Finally, one limitation of ZipperChain is a lack censorship resistance in that its infrastructure is centralized. However, any organization or individual is able to create its own instance of ZipperChain to process their own transaction and provably demonstrate which of these are on the main chain.

6 Related work

Early blockchain designs, such as Bitcoin [33], were made possible by a Proof-of-Work (PoW) leading to the Nakamoto probabilistic consensus. A miner creates a new block by solving a cryptographic puzzle and guess a *nonce*, the hash of which, together with other parts of the blockchain, produces a hash that sufficiently small, when considered as a binary number. While this mechanism has proven resilient to coordinated attacks, it is costly in terms electricity used by mining hardware. To address the cost of mining of new blocks, Peercoin [27] was the first to adopt Proof-of-Stake (PoS), where the opportunity to create the next block is decided by a lottery weighted by the number of coins staked by a verifier node, rather than the node's hash power. The correctness of the lottery is usually enforced by a Byzantine Fault Tolerant (BFT) consensus mechanism. Both PoW and PoS designs have led to a number of well-established public blockchains [2, 33].

The limiting factor to the performance of these blockchains is the network performance between its miner/verifier nodes [28]. It simply takes some time to disseminate a new block, so that the verifiers can create its successor, rather than a fork. The block interval then is governed by the size of the block, block interval, and network performance. While one might naturally worry about block processing time, for example in face of complex smart contracts, verifier processing speed has not been the limiting factor to blockchain performance as of yet [25, 28].

To gain higher performance DLT design departs in two directions. The first of these aims, primarily, to reduce transaction delay by decreasing block interval by using smaller blocks that take less time to disseminate. The second aims, primarily, to increase transaction throughput by relying on a constrained number of well-connected verifier nodes. Some blockchains combine the two approaches [8, 35].

When decreasing block interval ad infinitum blocks become so small as to contain only a single, or a few transaction, but reference more than one previous blocks to form a directed acyclic graph (DAG) [8, 21, 35, 39]. In a DAG blocks at the same height in the DLT do not necessarily create a fork, which allows nodes to create them independently and tie forks together with subsequent blocks. Transaction finality still depends on agreement among some quorum of nodes, but nodes reach agreement independently on each transaction, which tends to speed up transaction delay, but not necessarily throughput. A good example is Nano with the impressive transaction delay of 0.146 s, but a relatively measly throughput of 1,202 tps [3].

The second direction, when constraining the number of verifiers, the speed of block dissemination improves through high capacity, direct connections between verifiers. A DLT may identify these verifiers through delegation as in Nano [8], or by configuring their identities into the protocol in a scheme dubbed proof-of-authority (PoA) as in the roll out of Polkadot [10]. In addition to offering higher throughput and lower latency between nodes, PoA networks consume less power than PoW, are potentially more secure than PoS since verifier identities are known [16]. On the other hand the small number of nodes makes PoA networks vulnerable to 51% and distributed denial of service (DDoS) attacks, and their users to censorship and blacklisting [14]. Not following the protocol forfeits validator reputation and with it, the right to make future blocks. Unfortunately, in most systems, it is not clear how to quantify reputation.

7 Conclusions

BLOCKY ZipperChain guarantees immutability, agreement, and availability of transaction data, but without relying on distributed consensus. Instead, its construction process transfers trust from widely-used, third-party services onto ZipperChain's correctness guarantees. ZipperChain blocks are built by a pipeline of specialized services deployed on a small number of nodes connected by a fast data center network. As a result, ZipperChain transaction throughput approaches network line speeds. Our goal is to support 1 million transactions per second with block finality of 100 ms. Finally, ZipperChain infrastructure creates blocks centrally and so does not need a native token to incentivize a community of verifiers.

References

[1] SEC Interpretation:Electronic Storage of Broker-Dealer Records. https://www.sec.gov/rules/interp/34-47806.htm, May 2003.

[2] EOS.IO Technical White Paper v2. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md#consensus-algorithm-bft-dpos, April 2018.

[3] Nano Stress Tests - Measuring BPS, CPS, & TPS in the real world. https://forum.nano.org/t/nano-stress-tests-measuring-bps-cps-tps-in-the-real-world/436, August 2020.

[4] Amazon Cognito: User Guide. https://docs.aws.amazon.com/pdfs/cognito/latest/developerguide/cognito-dg.pdf, December 2022.

[5] AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/, March 2022.

[6] AWS Nitro Enclaves Root Certificate. https://aws-nitro-enclaves.amazonaws.com/AWS_NitroEnclaves_Root-G1.zip, Accessed March 2022.

[7] AWS Nitro Enclaves User Guide. https://docs.aws.amazon.com/enclaves/latest/user/enclaves-user.pdf, Accessed February 2022.

[8] Nano - Digital money for the modern world. https://docs.nano.org/living-whitepaper/, Accessed April 2022.

[9] Nitro Secure Module library. https://github.com/aws/aws-nitro-enclaves-nsm-api, February 2022.

[10] Polkadot Launch: Proof of Authority. https://polkadot.network/launch-poa/, Accessed April 2022.

[11] The Security Design ofthe AWS Nitro System: AWS Whitepaper. https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf, November 2022.

[12] Zipperchain durability analysis. https://tinyurl.com/5u6cb5n9, April 2022.

[13] Amazon. Amazon Simple Storage Service: User Guide. https://docs.aws.amazon.com/pdfs/AmazonS3/latest/userguide/s3-userguide.pdf, November 2022.

[14] Binance Academy. Proof of Authority Explained. https://academy.binance.com/en/articles/proof-of-authority-explained, December 2020.

[15] BLOCKY. ZipperChain Replication Service. https://colab.research.google.com/drive/1ZapogFjC314uE2kQfooSUj4fvzIqDHUU, October 2022.

[16] Dimitar Bogdanov. Proof of Authority Explained. https://limechain.tech/blog/proof-of-authority-explained/, August 2021.

[17] Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, January 2012.

[18] Sébastien Briais and Uwe Nestmann. A formal semantics for protocol narrations. In *Trustworthy Global Computing*, April 2005.

[19] Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. EIP-1559: Fee market change for ETH 1.0 chain. https://eips.ethereum.org/EIPS/eip-1559, April 2019.

[20] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A Platform for Confidentiality-Preserving,Trustworthy, and Performant Smart Contracts. https://arxiv.org/pdf/1804.05141.pdf, August 2019.

[21] Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. In *ACM Advances in Financial Technologies*, October 2019.

[22] Google. How Cloud Storage delivers 11 nines of durability. https://cloud.google.com/blog/products/storage-data-transfer/understanding-cloud-storage-11-9s-durability-target, January 2021.

[23] Google. Retention policies and retention policy locks. https://cloud.google.com/storage/docs/bucket-lock, March 2022.

[24] D. Hardt. The OAuth 2.0 Authorization Framework. https://datatracker.ietf.org/doc/html/rfc6749, October 2012.

[25] Alexander Hentschel, Dieter Shirley, and Layne Lafrance. Flow: Separating Consensus and Compute. https://arxiv.org/pdf/1909.05821.pdf, September 2019.

[26] T. Ho, M. Medard, R. Koetter, D.R. Karger, M. Effros, J. Shi, and B. Leong. A Random Linear Network Coding Approach to Multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430, September 2006.

[27] Sunny King and Scott Nadal. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. https://www.peercoin.net/read/papers/peercoin-paper.pdf, August 2012.

[28] Uri Klarman, Soumya Basu, Aleksandar Kuzmanovic, and Emin Gun Sirer. bloXroute: A Scalable Trustless Blockchain Distribution Network. https://bloxroute.com/wp-content/uploads/2018/03/bloXroute-whitepaper.pdf, March 2018.

[29] Jae Kwon and Ethan Buchman. Cosmos Whitepaper. https://v1.cosmos.network/resources/whitepaper, Accessed March 2022.

[30] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. https://datatracker.ietf.org/doc/html/rfc4122, July 2005.

[31] Microsoft. Legal holds for immutable blob data. https://docs.microsoft.com/en-us/azure/storage/blobs/immutable-legal-hold-overview, December 2021.

[32] Microsoft. Azure Storage redundancy. https://docs.microsoft.com/en-us/azure/storage/common/storage-redundancy, March 2022.

[33] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[34] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Winter Usenix Conference*, February 1998.

[35] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and Probabilistic Leaderless BFT Consensus through Metastability. https://arxiv.org/abs/1906.08936, June 2019.

[36] Brian Wilson. Backblaze Durability Calculates at 99.999999999Doesn't Matter. https://www.backblaze.com/blog/cloud-storage-durability/, July 2018.

[37] Gavin Wood. Polkadot: Vision For A Heterogeneous Multi-Chain Framework. https://polkadot.network/PolkaDotPaper.pdf, October 2016.

[38] Anatoly Yakovenko. Inside Solana's Internal Scalability Test. https://medium.com/solana-labs/inside-solanas-internal-scalability-test-cce13aa8c859, October 2019.

[39] Anatoly Yakovenko. Solana: A new architecture for a high performance blockchain v0.8.13. https://solana.com/solana-whitepaper.pdf, Accessed April 2022.

[40] Guy Zyskind, Oz Nathan, and Alex 'Sandy' Pentland. Enigma: Decentralized Computation Platform withGuaranteed Privacy. http://livinglab.mit.edu/wp-content/uploads/2016/01/enigma_full.pdf, January 2016.