# Setchain Algorithms for Blockchain Scalability*

Arivarasan Karmegam[a,c], Gabina Luz Bianchi[f], Margarita Capretto[b,d], Martín Ceresa[e], Antonio Fernández Anta[b,a], César Sánchez[b]

[a] *IMDEA Networks Institute, Madrid, Spain*
[b] *IMDEA Software Institute, Madrid, Spain*
[c] *Universidad Carlos III de Madrid, Madrid, Spain*
[d] *Universidad Politécnica de Madrid, Madrid, Spain*
[e] *Input-Output, Madrid, Spain*
[f] *Universidad Nacional de Rosario, Rosario, Spain*

**Abstract**

Blockchain scalability remains a longstanding obstacle to its broader adoption. To mitigate this limitation, numerous approaches have been proposed to improve blockchain throughput and efficiency. One such approach is Setchain, a reliable distributed object that improves scalability by relaxing the strict total-order requirement among transactions. Setchain arranges elements as an ordered sequence of sets, called *epochs*. Elements inside an epoch have no internal order, while epochs themselves are totally ordered.

In this work, we introduce and evaluate three Setchain algorithms built on top of a block-based ledger. *Vanilla* is a straightforward baseline implementation. *Compresschain* groups elements into batches and compresses them before appending them as epochs on the ledger. *Hashchain* instead maps each batch to a fixed-length hash, which is then appended as an epoch in the ledger. This design requires Hashchain to rely on a distributed service to retrieve the batch contents corresponding to a given hash.

To enable light clients to safely interact with a single server, the proposed algorithms store epoch-proofs within the Setchain. An *epoch-proof* is the hash of the epoch, cryptographically signed by a server. A client can verify the correctness of an epoch with $f+1$ epoch-proofs, where $f$ is the maximum

*This work is funded in part by a research grant from Nomadic Labs and the Tezos Foundation, and by MICIU/AEI /10.13039/501100011033/, ERDF, and the ESF+ under predoctoral training grant PREP2022-000373, grant DRONAC (PID2022-140560OB-I00), grant DECO (PID2022-138072OB-I00), and grant CEX2024-001471-M.

number of Byzantine servers assumed. We implement all three Setchain variants on the CometBFT blockchain application platform and deploy them as Docker-based nodes in a cluster. We evaluate performance under multiple settings using clusters with four, seven, and ten servers. The results indicate that the Setchain algorithms achieve throughput orders of magnitude higher than that of the underlying blockchain, while reaching finality with latency under 4 seconds.

*Keywords:*
Blockchain, Setchain, Scalability, CometBFT

---

## 1. Introduction

The first blockchain deployed was designed by Satoshi Nakamoto [35], building on previous proposals [17, 29], as a method to eliminate trusted third parties in the implementation of the Bitcoin cryptocurrency. Modern blockchains incorporate smart contracts [46, 50], which are immutable stateful programs stored in blockchains that extend the functionality of transactions beyond the exchange of cryptocurrency. Smart contracts allow the implementation of reliably decentralized sophisticated functionality, for example, enabling many applications in finances (DeFi) [4], governance (DAO) [40] and computation (Web3) [13].

A blockchain is a *reliable distributed object* that stores the sequence of transactions executed on behalf of users and the transactions are totally ordered and packed into blocks [22, 21]. Real-world blockchains are maintained by multiple servers without a central authority, and they rely on a *Byzantine-tolerant consensus algorithm* to establish transaction order and determine the resulting state updates [16, 18].

### *1.1. Challenges*

A key challenge for real-world blockchains is appending transactions to the ledger at a high rate. This rate is known as *throughput* and is typically expressed in transactions per second (TPS). Throughput depends on several bottlenecks, including the speed with which consensus is reached, network latency, and the computational overhead of executing transactions to update or validate the blockchain state. To preserve efficiency and decentralization, many systems [35, 13] also enforce limits on block size or execution complexity, which can further restrict throughput.

A second challenge is lowering blockchain *latency*, i.e., the time between a user submitting a transaction and that transaction being included in the blockchain. In some blockchains, this latency can reach hours [49, 35].

Moreover, in some blockchains (e.g., Bitcoin), a block that is initially reported as appended to the blockchain may later be discarded due to a chain fork. Thus, a third challenge is achieving *finality*, namely identifying a point after which users can be confident that their transactions are permanently recorded and will not be removed.

Finally, a fourth challenge is providing users with *proofs* that their transactions are indeed in the chain. This can be done by having users contact multiple servers and request individual proofs (e.g., a full copy of the chain). In practice, this is rarely done; instead, users commonly trust that the single server they interact with is honest. It is therefore desirable to provide users (who contact only one server) with strong guarantees that their transactions have been appended to the chain.

We address these challenges with Setchain [14], which is a *reliable distributed object* that implements Byzantine-tolerant distributed grown-only sets with barriers. Setchain has been shown to have a large throughput, and it uses *epoch-proofs,* which are cryptographic artifacts that allow users to interact reliably with only one server.

### *1.2. Related Work*

Scalability remains a central challenge as blockchains gain popularity. Various techniques have been proposed to address this issue. Bitcoin, one of the most widely known blockchains, operates on Proof-of-Work (PoW), offering robust security and decentralization. However, Bitcoin suffers from low throughput, averaging around 7 TPS, due to its block size and 10-minute block intervals [47]. Additionally, Bitcoin's latency is significant, with blocks typically taking 10 minutes to confirm, and (optimistic) finality is reached after about 60 minutes with the six-block confirmation rule [49].

Ethereum, which now uses PoS, handled 12-30 TPS [42] with PoW. Finality is generally reached within 12 to 15 minutes, as two "epochs" (each 32 slots or 6.4 minutes) are justified and finalized [9]. Sharding [33, 52], is an on-chain solution which partitions the blockchain into smaller segments to enable parallel transaction processing. Though "The Merge" did not improve Ethereum's throughput, as its main goal was to transition to PoS, Ethereum's Danksharding initiative, part of the larger Ethereum 2.0

roadmap, is projected to increase throughput to more than 100,000 TPS once fully implemented [1].

While sharding is still under development, Layer 2 solutions like rollups provide immediate scalability benefits. Layer 2 solutions include state channels [39], sidechains, and various forms of rollups, such as Plasma [38], Optimistic Rollups [31], and Zero Knowledge Rollups [2]. These technologies facilitate high-volume transaction processing off the main blockchain, thereby alleviating network congestion and enhancing transaction speed and cost efficiency.

Arbitrum [31], a Layer 2 solution for Ethereum, enhances scalability using Optimistic Rollups. Its recent upgrade, Arbitrum Nitro, theoretically enables the network to achieve up to 40,000 TPS. However, transaction latency and finality on Arbitrum depend on Ethereum Layer-1 latency and finality, as Arbitrum posts batches of transactions to Ethereum for settlement. Arbitrum employs a fraud-proof mechanism with a one-week resolution period, though such challenges are uncommon.

Algorand [24], using a Pure Proof-of-Stake (PPoS) consensus mechanism, offers a throughput of around 6,000 TPS. It provides fast finality, with transactions confirmed in approximately 3.5 seconds. Cosmos [48], which employs the Tendermint consensus algorithm [11], can handle around 10,000 TPS. Tendermint provides finality within 6-7 seconds. However, the performance depends on the specific blockchain's design, optimizations, and network conditions.

Hyperledger Fabric, a permissioned blockchain, introduced Byzantine Fault Tolerant (BFT) ordering service in version v3.0.0 [20] through the SmartBFT consensus library [6], enabling Fabric to handle Byzantine faults. Fabric typically achieves thousands of TPS depending on the configuration, with fast finality upon block commitment. However, the addition of BFT may slightly increase latency due to added consensus steps.

Another approach to scaling blockchain is consensus optimization. By implementing a Set Byzantine Consensus (SBC), Red Belly [18] enhances the scalability, allowing the network to agree on a superblock of all proposed blocks. RedBelly demonstrates impressive scalability in testing environments, achieving over 660,000 TPS and up to 30,000 TPS in real-world conditions. Fast finality ensures that transactions are final and irreversible within 3 seconds, further enhancing its appeal for high-throughput applications. Algorithms for SBC or that use SBC are an active area of research [18, 14, 41], as they exploit the lack of order between transactions.

Table 1: Comparison of blockchain scalability approaches. †Analytical throughput bound for Hashchain with collector size $c = 500$; see Section 7.1.

| System | Ordering | Throughput | Finality |
|---|---|---|---|
| Bitcoin | Total | ∼7 TPS | ∼60 min |
| Ethereum (PoS) | Total | 12–30 TPS | 12–15 min |
| Algorand | Total | ∼6,000 TPS | ∼3.5 s |
| Cosmos | Total | ∼10,000 TPS | 6–7 s |
| RedBelly | Total | 30,000–660,000 TPS | ∼3 s |
| Narwhal + Tusk | Total | ∼170,000 TPS | ∼3 s |
| Bullshark | Total | ∼125,000 TPS | ∼2s |
| Hyperledger Fabric | Total | Thousands TPS (config-dependent) | Deterministic upon block commit |
| **Setchain** | **Epoch-ordered sets** | **∼147,857 TPS†** | **<4s** |

Hotstuff [51] is a leader-based BFT replication consensus protocol with linear communication complexity and optimistic responsiveness. The Hotstuff paper proposes a variant of the protocol, Chained HotStuff, which is a pipelined version of Basic HotStuff in which a Quorum Certificate can serve in multiple phases simultaneously. There are variants of this work that are proposed to improve its performance. For example, P-Hotstuff [53], which introduces parallelism into Hotstuff, claims to achieve an average throughput 20 times higher than that of Hotstuff.

FireLedger [12] is a BFT consensus protocol designed to optimize throughput and latency in permissioned blockchain environments by leveraging the iterative nature of blockchains to improve their throughput in optimistic execution scenarios. In their evaluation, the authors report that FireLedger, when deployed on ten mid-range Amazon instances within a single data center, can reach a throughput of approximately $160,000$ TPS for transactions of size 512 bytes. In a geo-distributed deployment across ten Amazon nodes, the protocol achieves around $30,000$ TPS for the same transaction size.

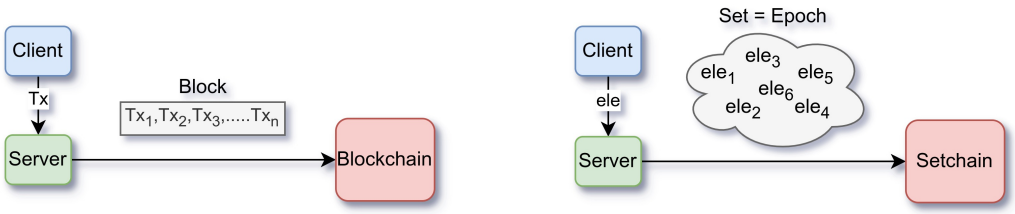
Dumbo [28] is an improvement on Honeybadger [34] (the first practical asynchronous BFT protocol). Dumbo replaces HoneyBadger's $n$ concurrent Asynchronous Binary Agreements (ABAs) with a single Multi-valued Validated Byzantine Agreement (MVBA), improving both latency and throughput. Dumbo-NG [28] builds on this idea by decoupling transaction dissemination from agreement and executing them concurrently. It uses a bandwidth-oblivious MVBA with threshold signatures, achieving 4–8× higher peak throughput and a latency that remains stable as throughput increases.

Narwhal [19] decouples reliable transaction dissemination from ordering via a DAG-based mempool that certifies availability and supports bounded-memory garbage collection and scale-out workers. When paired with Hot-Stuff (Narwhal-HS), it sustains approximately 130–140, 000 TPS with less than 2 seconds of latency in WAN settings with 50 validators, and adding workers scales throughput roughly linearly to 600, 000 TPS without increasing latency. Tusk adds the actual consensus on top of Narwhal: a fully asynchronous, zero-message-overhead protocol that orders by interpreting the local DAG with a shared random coin. In WAN experiments, it achieves 170, 000 TPS at 3 seconds with 50 validators, and under 1 and 3 crash faults, it keeps latency below 4 and 6 seconds while maintaining high throughput.

Bullshark [44] is a DAG-based Byzantine atomic broadcast protocol optimized for the common (partially synchronous) case: it adds a fast path that commits in 2 rounds during synchrony while retaining an asynchronous fallback with 6-round expected latency and $O(n)$ amortized communication. Built atop Narwhal[19], it decouples data dissemination from ordering, avoids view-change or synchronization, and provides timely fairness with bounded memory via garbage collection. In evaluation, a partially synchronous Bullshark variant reaches around 125, 000 TPS at 2 second latency with 50 parties.

Fantastyc [10] proposes a byzantine-tolerant design that anchors only cryptographic proofs on-chain while keeping client updates and aggregated models off-chain in a fault-tolerant key-value store keyed by the data's hash. A server collects, updates, stores key-values `(hash(v), v)` off-chain, and, after gathering $f + 1$ signed hash-keys, produces a Proof of Availability & Integrity (PoA&I). The blockchain records sets of PoA&I (typically one transaction per round) rather than raw data, and clients later retrieve and verify values directly from the store using the anchored hashes. This cleanly decouples ordering (chain), integrity (servers), and availability (storage) and allows light clients to validate using PoA&I alone. Conceptually, this mirrors Hashchain's approach of appending hashes of batches on the ledger and consolidating an epoch only after $f + 1$ signatures on the batch hash, to ensure at least one honest replica can serve the data, thereby slashing on-chain bandwidth while preserving retrievability.

Table 1 summarizes the key properties of the approaches discussed in this section.




(a) Blockchain (b) Setchain

Figure 1: Blockchain vs Setchain

### *1.3. Setchain*

A promising approach to enhance blockchain scalability is Setchain [14], a Byzantine-tolerant distributed object that implements a sequence of sets (called *epochs*). By relaxing the strict total-order constraint of conventional blockchains (See Figure 1), Setchain can improve throughput and scalability by enabling transactions within the same epoch to be validated in parallel. Setchain fits applications such as digital registries (e.g., MIT Digital Diplomas [8, 37], Georgia Government registry [26]) or voting systems (e.g., Follow My Vote [23], Chirotonia [43]), where elements do not require a total order except at infrequent epoch barriers. Although Byzantine-tolerant distributed algorithms that implement Setchain have been proposed, efficient real-world implementations are still lacking.

### *1.4. Contributions*

In this work, we propose a family of real-world Setchain algorithms built on top of a block-based ledger. We adopt an incremental methodology, introducing a sequence of increasingly refined approximations that lead to the final (and most sophisticated) solution (Section 3). We begin with **Algorithm Vanilla**, a straightforward realization of Setchain whose throughput is comparable to that of the underlying ledger. We then introduce **Algorithm Compresschain**, which improves throughput by compressing epochs. Finally, we present **Algorithm Hashchain**, which relies on hash functions.

Among these algorithms, our main contribution is Hashchain. Hashchain leverages the succinctness of hashes to reduce the communication required during broadcasts and consensus, sending a fixed-size hash rather than the hundreds or thousands of elements that may comprise an epoch. The trade-off is the need for an additional distributed algorithm to retrieve the epoch

contents associated with a hash from the corresponding server.

In addition, to enable users to interact with only one server, these algorithms maintain epoch proofs as part of the Setchain. An *epoch-proof* is the hash of an epoch, cryptographically signed by a server. A user can validate the correctness of an epoch using $f + 1$ epoch-proofs (where $f$ is the maximum number of Byzantine servers).

We implement all three Setchain algorithms on top of the CometBFT blockchain application platform (Section 6) and deploy them as Docker nodes in a cluster. We evaluate performance under a range of configurations using clusters of four, seven, and ten servers. Our results show that the Setchain algorithms achieve throughput orders of magnitude higher than that of the underlying blockchain (tens of thousands of TPS) and provide finality with latency below 4 seconds.

### *1.5. Structure*

The rest of the article is structured as follows: In Section 2, we present a brief description of our system model, Setchain, and other relevant concepts. Section 3 describes three implementations of Setchain on top of a block-based ledger. In Section 4, we present the proof of correctness for the Setchain algorithms. In Section 5, we present an analytical study of the throughput of the Setchain algorithms. In Section 6, we describe the implementation details. Finally, in Section 7, we present and discuss our results and Section 8 concludes.

This article is an extended version of our conference paper [32]. Compared to the conference version, we include a detailed related-work discussion (Section 1.2), algorithmic details that were omitted due to space constraints (for example, Algorithm Vanilla), an analytical performance study of the Setchain Algorithms (Section 5), and complete correctness proofs for all proposed Setchain Algorithms (Section 4). We also provide additional experimental/analytical details and discussion to support a deeper understanding of the performance results.

## 2. Model and Definitions

### *2.1. System Model*

Table 2 summarizes the notation used throughout the paper. We consider a distributed system comprising $n$ servers and an unbounded number

Table 2: Summary of notations used throughout the paper.

| Symbol | Description |
|---|---|
| $n$ | Total number of servers |
| $f$ | Maximum number of Byzantine servers |
| $e$ | A setchain element |
| $U$ | Universe of valid elements |
| $tx$ | A transaction in the ledger |
| `L` | The block-based ledger object |
| `S` | The Setchain object |
| $S_i$ | Server $i$, for $i \in \{1, \ldots, n\}$ |
| $G$ | Batch of valid, unassigned elements forming a new epoch |
| $B$ | A block delivered by the ledger |
| $ep$ | Epoch-proof of server $S$: $\langle epoch, p, S \rangle$ |
| $epoch$ | epoch number |
| $p$ | $\texttt{Sign}_S(\texttt{Hash}(epoch, G))$ |
| $hb_i$ | Hash-batch tuple of server $S_i$: $\langle h, s_i, S_i \rangle$ |
| $s_i$ | Signature of server $S_i$ |
| $h$ | Hash of a batch: $h = \texttt{Hash}(batch)$ |
| $C$ | Block capacity in bytes |
| $c$ | Collector size (max elements per batch) |
| $r$ | Compression ratio (Compresschain) |
| $R$ | Ledger block production rate (blocks/s) |
| $T_v, T_c, T_h$ | Analytical throughput of Vanilla, Compresschain and Hashchain (TPS) |
| $l_e, l_p, l_h$ | Size of one element, an epoch-proof and a hash batch (in bytes) |

of clients; together, they constitute the system processes. The system is permissioned yet public (also known as "open permissioned" [18]), referring to an open model for clients, but where servers are known upfront (permissioned). This model can also be extended to a permissionless setting via committee sortition [25], without requiring substantial changes. Up to $f < n/2$ of the servers may exhibit Byzantine behavior, while clients are unrestricted (i.e., they may also be Byzantine). We assume that the value of $f$ is known. We use $f + 1$ as a lower bound on the number of consistent epoch-proofs needed to guarantee that an epoch is correct, and also as the number of signatures required to consolidate hashes into epochs (see Section 3.3).

We assume a public key infrastructure (PKI) in which each process (server or client) possesses a public-private key pair. Every process is assumed to know the public keys of all other processes. The communication between processes is reliable in the sense that messages sent between correct processes are eventually delivered exactly once, and no spurious messages are created.

In addition, beyond the requirements of the underlying building blocks, we assume synchronous communication between correct processes: messages are delivered within a bounded time. This assumption is used by timeout-based mechanisms in our algorithms, including but not limited to batch retrieval in Hashchain. We do not assume bounds on message size or bandwidth. Faulty processes may send arbitrary messages; however, due to the PKI, they cannot impersonate other processes. The PKI, therefore, provides authenticated and secure communication between processes. Each message is signed by its sender using their private key, and the receiver verifies the signature using the sender's public key. Messages with invalid signatures are discarded.

Clients create elements and invoke a Setchain operation to add them. Client-generated elements are signed and can hence be authenticated. Servers can also validate these elements for syntactic and semantic correctness. Correct servers process only authenticated, valid elements. We assume that a server cannot create a valid element on its own, and that clients and servers do not collude.

### *2.2. Setchain*

A Setchain [14] is a Byzantine-tolerant distributed object `S` that implements a sequence of sets called *epochs*. The Setchain imposes an order across different epochs, but does not order elements within the same epoch. Thus, by relaxing the total-order constraint of traditional blockchains, Setchain can potentially deliver higher throughput and improved scalability.

#### *2.2.1. Setchain API*

Let $U$ be the set of valid elements that client processes may inject into the Setchain. A Setchain `S` is a reliable distributed object in which a set of servers maintain:

- a grow-only set $the\_set \subseteq U$ containing all the elements that have been added;
- a natural number $epoch \in \mathbb{N}$;
- a map $history : \{1, \ldots, epoch\} \to 2^U$ that associates each epoch number with the set of elements (subset of $the\_set$) stamped with a given epoch number[1];

[1] $2^U$ denotes the power set of $U$.

- a set *proofs* of epoch-proofs (see Section 2.3)[2].

Servers support two operations: `add` and `get`. A client requests a server $v$ to add an element $e$ to the Setchain `S` by invoking $\texttt{S.add}_v(e)$, whereas $\texttt{S.get}_v()$ returns the tuple (*the_set*, *history*, *epoch*, *proofs*) that server $v$ maintains. Servers may locally evaluate validity using a predicate $\texttt{valid\_element}(e)$, which decides whether an element $e$ is valid. In a Setchain, when a new epoch is created, servers jointly determine which added elements in *the_set* that have not yet been assigned an epoch are included in the next epoch, and then advance the epoch number. We refer to this procedure as an *epoch increment*. For the remainder of the paper, we assume that from any point in time there will be a future epoch increment. This is a reasonable assumption in practice, and can be enforced reliably through the use of timeouts.

From the client's perspective, a typical workflow proceeds as follows: a client invokes $\texttt{S.add}_v(e)$ at some server $v$ to append a new valid element $e$ in the Setchain. The element $e$ is disseminated among servers, and upon the next epoch increment, the servers attempt to include it in the newly created epoch. After waiting for some time, the client calls $\texttt{S.get}_w()$ at a (possibly different) server $w$ to confirm that the element has been effectively added to the Setchain and included in an epoch.

*2.2.2. Setchain Basic Properties*

Setchain implementations must satisfy certain properties that guarantee consistency between correct servers and that the valid elements added are eventually included in an epoch [14]. These properties hold only on correct servers, since Byzantine servers do not provide any guarantees.

**Property 1** (Consistent-Sets)**.** *Let* $(T, H, h, P) = \texttt{S.get}_v()$ *be the result of an invocation to a correct server* $v$*. Then, for all* $i \in \{1, \ldots, h\}, H[i] \subseteq T$*.*

**Property 2** (Add-Get-Local)**.** *Let* $\texttt{S.add}_v(e)$ *be an operation invoked in a correct server* $v$ *and let* $e$ *be valid. Then, eventually all invocations* $(T, H, h, P) = \texttt{S.get}_v()$ *satisfy* $e \in T$*.*

**Property 3** (Get-Global)**.** *Let* $v$ *and* $w$ *be two correct servers, let* $e$ *be a valid element, and let* $(T, H, h, P) = \texttt{S.get}_v()$*. If* $e \in T$*, then eventually all invocations* $(T', H', h', P') = \texttt{S.get}_w()$ *satisfy that* $e \in T'$*.*

[2]This set was not part of the API described in the original Setchain paper [14] and is one of the contributions of this work.

**Property 4** (Eventual-Get)**.** *Let $v$ be a correct server, let $e$ be a valid element and let $(T, H, h, P) = \texttt{S.get}_v()$. If $e \in T$, then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $e \in H'$.*

**Property 5** (Unique-Epoch)**.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and let $i, i' \in \{1, \ldots, h\}$ with $i \neq i'$. Then, $H[i] \cap H[i'] = \emptyset$.*

**Property 6** (Consistent-Gets)**.** *Let $v, w$ be correct servers, $(T, H, h, P) = \texttt{S.get}_v()$, $(T', H', h', P') = \texttt{S.get}_w()$, and $i \in \{1, \ldots, \min(h, h')\}$. Then $H[i] = H'[i]$.*

**Property 7** (Add-before-Get)**.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and $e \in T$ be a valid element. Then there was an operation $\texttt{S.add}_w(e)$ invoked in the past in some server $w$.*

Properties 1, 5, 6, and 7 are safety properties. Properties 2, 3, and 4 are liveness properties.

### *2.3. Setchain Epoch-proofs*

The original Setchain [14] proposal assumes that a client interacts with a correct server. In practice, however, clients do not know whether the server it contacts is correct or Byzantine. As a result, a client may need to query a sufficiently large number of servers (at least $f + 1$) to guarantee that at least one of them is correct. In this work, we introduce *epoch-proofs* as a mechanism that enables clients to achieve the same guarantees without contacting multiple servers, simplifying the process and improving efficiency.

#### *2.3.1. Definition*

An epoch-proof is the cryptographic signature of an epoch $i$ by a server $v$. As discussed above, in this work, the Setchain maintains a set *proofs* of epoch-proofs, which is returned when a `get` operation is invoked. Therefore, if *proofs* contain at least $f + 1$ consistent epoch-proofs for a given epoch $i$, the client can be confident that the epoch is correct (See Figure 2). An epoch-proof is computed by signing the hash of the epoch number together with the epoch's elements: $ep_v(i) = \texttt{Sign}_v(\texttt{Hash}(i, \mathit{history}[i]))$.

#### *2.3.2. Usage*

To add an element, a client sends a single $\texttt{S.add}_v(e)$ request to one server $v$, with the hope that $v$ is a correct server. After waiting for some time, the

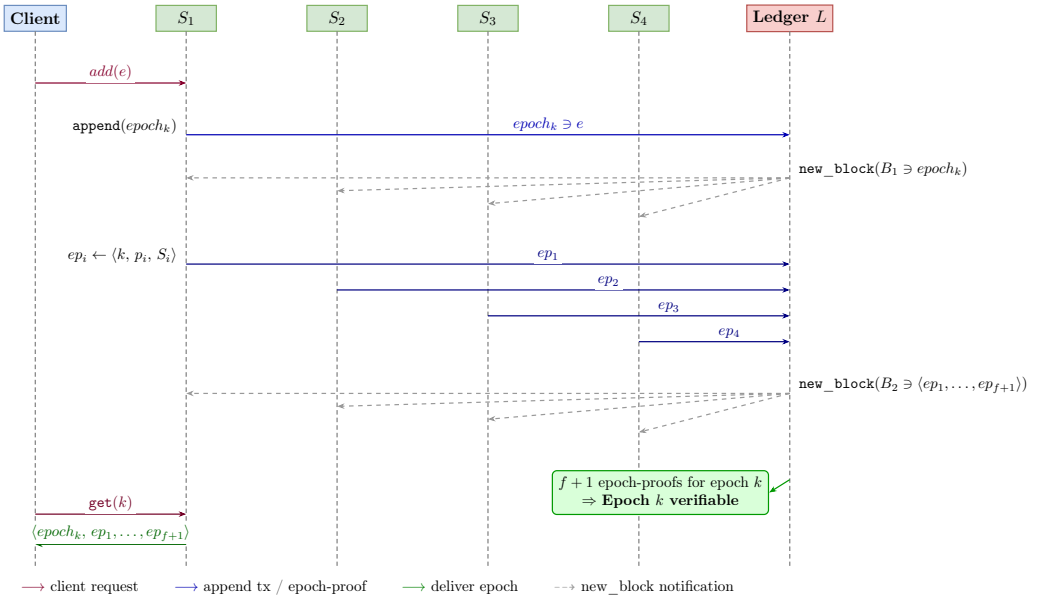



Figure 2: Epoch-proof life-cycle with $n = 4$ servers and $f = 1$. A client submits element $e$; the server forms $epoch_k \ni e$, appends it to the ledger, and upon block delivery, signs and appends their epoch-proofs $ep_i = \langle k, p_i, S_i \rangle$. Once $f + 1$ consistent epoch-proofs are recorded in the ledger through a new block, any correct server can serve the verified epoch to the client.

client can invoke a $\texttt{S.get}_w()$ from a single server $w$ and check whether $e$ is in some epoch, and whether the returned *proofs* set contains at least $f + 1$ valid epoch-proofs for that epoch, ensuring that at least one correct server has signed it. Clients validate epoch-proofs by recomputing the hash of the corresponding epoch and verifying the signature in the epoch-proof against that hash using the public key of the signing server. Recall that servers' public keys are known to clients. This procedure typically requires only one message per `add` and one message per `get`. It is possible that the client is unlucky; if servers $v$ or $w$ are Byzantine, then after waiting a reasonable amount of time, the client may need to restart the procedure. The following is the basic property that any Setchain algorithm must satisfy concerning epoch-proofs.

**Property 8** (Valid-Epoch)**.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and $i \in \{1, \ldots, h\}$. Then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $P'$ contains at least $f + 1$ epoch-proofs of $H[i]$.*

### *2.4. Block-based Ledger*

In the Setchain algorithms proposed in this paper, we assume access to a Byzantine-tolerant consensus service. For simplicity, we abstract this service as a block-based ledger. Although our algorithms only require that the number of Byzantine processes satisfies $f < n/2$, the underlying block-based ledger may impose stricter requirements. For example, in Section 6, we use the CometBFT, which requires $f < n/3$.

#### *2.4.1. Definition*

A block-based ledger `L` is a Byzantine-tolerant distributed object that maintains a sequence of blocks. Each block contains a sequence of transactions appended to the ledger by its clients. We prefer not to call this object a blockchain, since its transactions lack semantics. To keep the terminology consistent, we always use *transaction* to mean a *block-based ledger transaction*, whereas *element* refers to Setchain elements. Depending on the implementation, a single ledger transaction may contain one or multiple elements.

#### *2.4.2. API*

The block-based ledger provides two endpoints. First, `append`$(tx)$ is used to submit a transaction $tx$ to the ledger, which is eventually included in a block. Second, `new_block`$(B)$ notifies the servers whenever a new block $B$ has been appended. The number of transactions in $B$ is $|B|$, and the $i$-th transaction in $B$ is $B[i]$.

#### *2.4.3. Properties of the block-based ledger*

Since Setchain algorithms use a block-based ledger, proving correctness requires defining properties that the ledger guarantees. In particular, the ledger property used is that any valid transaction appended by a correct server will eventually be included in a block, which is then notified to all servers. Moreover, we assume that all blocks notified are final.

**Property 9.** *Ledger-Add-Eventual-Notify: Let $tx$ be a valid transaction, and let $v$ be a correct server. Then, if $v$ invokes* `L.append`$_v(tx)$*, transaction $tx$ will be eventually and permanently added in a fixed position $i$ within a block $B$. Moreover, all correct servers $w$ will be eventually notified of the new block $B$ with* `L.new_block`$_w(B)$.

**Property 10** (Ledger-Consistent-Notification)**.** *All correct servers $w$ are notified with* $\texttt{L.new\_block}_w(B)$ *of the same set of blocks, and in the same order.*

**Property 11** (Notification-Implies-Append)**.** *If a correct server $w$ is notified with* $\texttt{L.new\_block}_w(B)$ *containing a valid transaction $tx$, then some server $v$ had invoked* $\texttt{L.append}_v(tx)$*.*

## 3. Setchain Algorithms

This section presents practical Setchain algorithms built on top of a block-based ledger. We describe three implementations, starting from a simple baseline that is trivially correct and progressing to a more involved design that relies on hashing. As discussed in Section 2.2, a Setchain `S` provides two methods, `add` and `get`, which we instantiate for each algorithm.

### *3.1. Algorithm Vanilla*

Algorithm Vanilla (See Figure 3) is a basic implementation of Setchain `S` on top of a block-based ledger `L`. In Vanilla, each server $v$ maintains the local sets: *the_set*, *history*, and *proofs*, and the counter *epoch*, as defined in Section 2.2. When a client invokes $\texttt{S.get}_v()$, the server $v$ returns its current *the_set*, *history*, *epoch*, and *proofs* (Line 9).

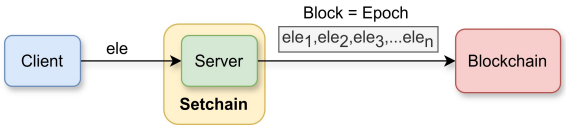



Figure 3: Algorithm Vanilla

Clients append to Setchain an element $e$ by invoking $\texttt{S.add}_v(e)$. The server only accepts valid elements not already present in *the_set* (see Line 4). If $e$ satisfies these conditions, the server adds $e$ to *the_set*, and calls $\texttt{L.append}_v(e)$, which eventually appends $e$ to ledger `L`.

Whenever a new block is appended to the ledger `L`, server $v$ is notified through $\texttt{L.new\_block}(B)$ (see Line 9). In all three algorithms, the ledger `L` is also used to disseminate epoch-proofs. Accordingly, upon receiving a block $B$, the server first extracts the valid epoch-proofs contained in $B$ and adds them to the set *proofs* (Line 11). Next, the server extracts from $B$ the valid, unassigned elements and collects them into a batch $G$ (validity checks remain necessary because Byzantine servers may append invalid elements to the ledger). The server adds $G$ to *the_set*, forms a new epoch from $G$, and

**Algorithm Vanilla** Code executed by server $v$. Uses block-based ledger `L` shared by all servers.

```
 1 Init: the_set ← ∅, epoch ← 0, history ← ∅, proofs ← ∅
 2 function add(e)
 3     assert valid_element(e) ∧ e ∉ the_set
 4     the_set ← the_set ∪ {e}
 5     L.append(e)
 6     return
 7 function get()
 8     return (the_set, history, epoch, proofs)
 9 upon (L.new_block(B)) do
10     np ← {ep ∈ B :
       ep = ⟨j, p, w⟩ is an epoch-proof ∧ valid_proof(j, p, w, history[j])}
11     proofs ← proofs ∪ np
12     G ← {e ∈ B : e is an element ∧ valid_element(e) ∧ e ∉ history}
13     the_set ← the_set ∪ G
14     epoch ← epoch + 1
15     history[epoch] ← G
16     p ← Sign_v(Hash(epoch, G))
17     L.append(⟨epoch, p, v⟩)
18 end upon
```

stores it in the *history* map. Finally, it generates an epoch-proof (including the epoch number, the epoch signature, and the server identity) and appends this proof to the ledger by invoking `L.append` (Line 17).

Algorithm Vanilla addresses the proof requirement through epoch-proofs, enabling a client to safely rely on a single correct server. However, the resulting Setchain inherits the throughput and latency of the underlying block-based ledger `L`.

### *3.2. Algorithm Compresschain*

Algorithm Compresschain (See Figure 4) improves throughput relative to Vanilla. In Algorithm Compresschain, each server uses a *collector* that accumulates client elements and server-generated epoch-proofs until a target collector size is reached (or a timeout fires). Then, the collected batch, corresponding to a new epoch, is compressed and appended to the ledger as a single transaction.

**Algorithm Compresschain** Code executed by server $v$. Uses block-based ledger `L` shared by all servers.

```
1  Init: the_set ← ∅, epoch ← 0, history ← ∅, proofs ← ∅, batch ← ∅
2  function add(e)
3      assert valid_element(e) ∧ (e ∉ the_set)
4      the_set ← the_set ∪ {e}
5      add_to_batch(e)
6      return
7  function add_to_batch(e)
8      batch ← batch ∪ {e}
9      return
10 function get()
11     return (the_set, history, epoch, proofs)
12 upon (isReady(batch)) do
13     assert batch ≠ ∅
14     b ← Compress(batch)
15     L.append(b)
16     batch ← ∅
17 end upon
18 upon (L.new_block(B)) do
19     for i = 1 to |B| do
20         batch_original ← Decompress(B[i])
21         if batch_original = ∅ then continue
22         np ← {ep ∈ batch_original : ep = ⟨j, p, w⟩ is an epoch-proof
                   ∧ valid_proof(j, p, w, history[j])}
23         proofs ← proofs ∪ np
24         G ← {e ∈ batch_original : e is an element ∧ valid_element(e)
                   ∧(e ∉ history)}
25         the_set ← the_set ∪ G
26         epoch ← epoch + 1
27         history[epoch] ← G
28         p ← Sign_v(Hash(epoch, G))
29         add_to_batch(⟨epoch, p, v⟩)
30 end upon
```

Algorithm Compresschain extends Algorithm Vanilla with an additional set *batch*, which collects client elements and epoch-proofs. When a client

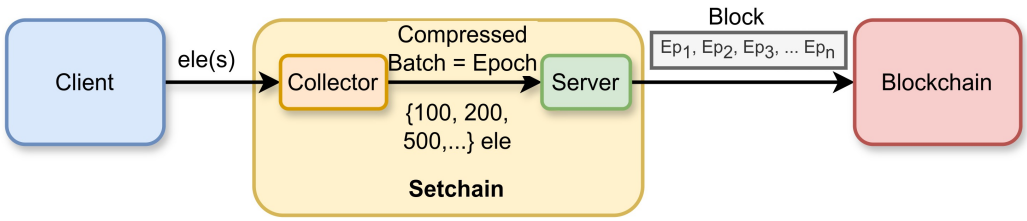



Figure 4: Algorithm Compresschain

submits an element $e$ via $\texttt{S.add}_v(e)$, the server adds $e$ to *the_set* and also enqueues it into *batch* using $\texttt{add\_to\_batch}(e)$ (Line 5). Once *batch* reaches the collector size (or a timeout is triggered while $batch \neq \emptyset$), the event $\texttt{isReady}(batch)$ is raised; the batch is then compressed, appended to the ledger, and *batch* is reset to $\emptyset$.

When a new block $B$ is delivered, the server processes each compressed batch in $B$ in order. Let $B[i]$ denote the $i$-th transaction of block $B$. The server decompresses $B[i]$, adds the valid epoch-proofs found in $B[i]$ to *proofs*, and extracts the valid elements of $B[i]$ into a set $G$ that will form a new epoch. The elements in $G$ are added to *the_set*, assigned an epoch number, and recorded in *history*. The server also creates an epoch-proof for the new epoch and forwards it to the collector via $\texttt{S.add\_to\_batch}_v(\cdot)$, which inserts it into *batch*.

The key difference between Algorithms Compresschain and Vanilla is the granularity of ledger transactions. In Algorithm Compresschain, each transaction in $B$ is a compressed batch (potentially containing many Setchain elements) that becomes an epoch, whereas in Algorithm Vanilla each transaction is a single element and an epoch corresponds to the set of valid elements in $B$. This change can substantially increase throughput.

### *3.3. Algorithm Hashchain*

Algorithm Hashchain (See Figure 5) increases the throughput by hashing batches instead of compressing them. While hashing can yield a substantial reduction in communicated size, hashes are not reversible; therefore, an additional mechanism is required to recover the original batch contents.

In Algorithm Hashchain, when a batch becomes ready (Line 12), server $v$ computes the batch hash $h$, signs $h$, and forms the hash-batch $hb$ by combining the hash, the signature, and the server identity. The server then appends $hb$ as a transaction to the ledger $\texttt{L}$ and resets *batch* to $\emptyset$.

**Algorithm Hashchain** Code executed by server $v$. Uses block-based ledger `L` shared by all servers.

```
1  Init: the_set ← ∅, epoch ← 0, history ← ∅, proofs ← ∅, hash_to_batch ← ∅,
         batch ← ∅, hash_to_signers ← ∅
2  function add(e)
3      assert valid_element(e) ∧ (e ∉ the_set)
4      the_set ← the_set ∪ {e}
5      add_to_batch(e)
6      return
7  function add_to_batch(e)
8      batch ← batch ∪ {e}
9      return
10 function get()
11     return (the_set, history, epoch, proofs)
12 upon (isReady(batch)) do
13     assert batch ≠ ∅
14     h ← Hash(batch)
15     hash_to_batch[h] ← batch
16     Register_batch(h, batch)
17     s ← Sign_v(h)
18     hb ← ⟨h, s, v⟩
19     L.append(hb)
20     batch ← ∅
21 end upon
22 upon (L.new_block(B)) do
23     for i = 1 to |B| do
24         if (B[i] = ⟨h, s_w, w⟩)∧ valid_hash(h, s_w, w)∧ w ∉ hash_to_signers[h] then
25             batch_original ← hash_to_batch[h]
26             if batch_original = ∅ then                                  ▷ h is new
27                 batch_original ← Request_batch(h)
28                 if (batch_original ≠ ∅) ∧ (Hash(batch_original) = h) then   ▷ Found and correct
29                     hash_to_batch[h] ← batch_original
30                     Register_batch(h, batch_original)
31                     s_v ← Sign_v(h)
32                     hb ← ⟨h, s_v, v⟩
33                     L.append(hb)
34                     np ← {ep ∈ batch_original : ep = ⟨j, p, w⟩ is epoch-proof ∧
                              valid_proof(j, p, w, history[j])}
35                     proofs ← proofs ∪ np
36                     G ← {e ∈ batch_original : e is an element ∧ valid_element(e) ∧(e ∉ history)}
37                     the_set ← the_set ∪ G
38             hash_to_signers[h] ← hash_to_signers[h] ∪{w}
39             if |hash_to_signers[h]| = f + 1 then
40                 epoch ← epoch + 1
41                 G ← {e ∈ hash_to_batch[h] : e is an element ∧ valid_element(e) ∧(e ∉ history)}
42                 history[epoch] ← G
43                 p ← Sign_v(Hash(epoch, G))
44                 add_to_batch(⟨epoch, p, v⟩)
45 end upon
```

Because the original batch cannot be reconstructed from $h$ alone, the

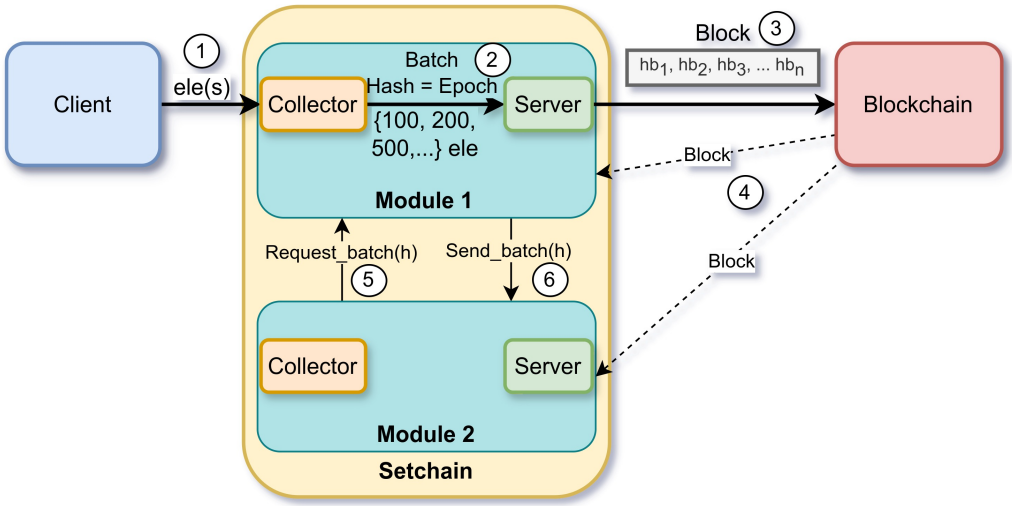



Figure 5: Algorithm Hashchain

server stores the batch associated with $h$ in the map *hash_to_batch*. In addition, the batch and its hash $h$ are registered via `Register_batch`, enabling other servers to request and obtain the batch contents.

Upon a `L.new_block`$(B)$ notification, the server processes each valid hash-batch $hb_w = \langle h, s_w, w\rangle \in B$ as follows. If server $v$ does not have the batch for $h$ in *hash_to_batch*, it sends `Request_batch`$(h)$ to server $w$, namely, to the signer of the hash-batch currently being processed. Since $w$ may be Byzantine, server $v$ waits only a bounded amount of time for a reply. If no batch is received, or if the received batch does not satisfy `Hash`$(batch\_original) = h$, then the batch is discarded and $v$ does not append its own hash-batch for $h$. Otherwise, $v$ stores the batch in *hash_to_batch*, updates *the_set*, creates its own hash-batch $hb_v = \langle h, s_v, v\rangle$, and appends it to the ledger `L`. Thus, a Byzantine server can delay retrieval from that signer, but it cannot make a correct server accept an invalid batch.

Observe that a hash-batch observed in ledger `L` cannot be immediately assigned an epoch number: a Byzantine server could append a hash-batch and then refuse to serve the corresponding batch. For this reason, we require that a hash be signed by at least $f + 1$ distinct servers before it can be consolidated into an epoch. Indeed, if hash-batches for the same $h$ exist from $f + 1$ different servers, then at least one of these signers is correct and will possess the batch and serve it upon request. Hence, selective withholding

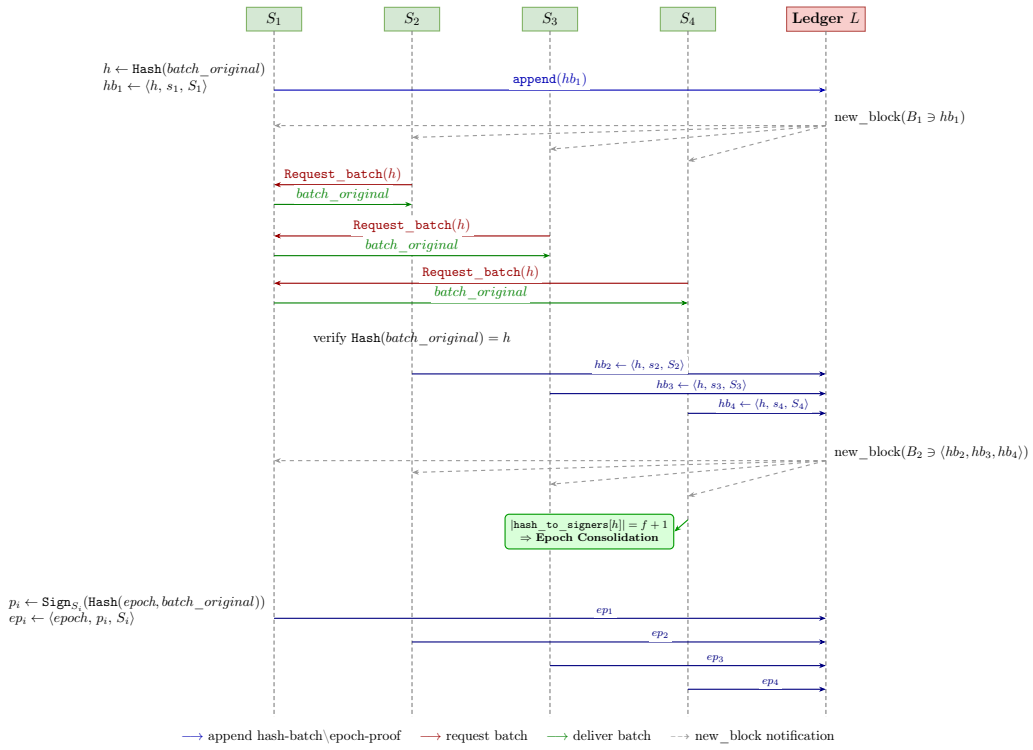



Figure 6: Hashchain batch retrieval and epoch consolidation with $n = 4$ servers and $f = 1$. Server $S_1$ appends hash-batch $hb_1$; the other servers retrieve the original batch, verify its hash, and append their own hash-batches. Once $f+1 = 2$ consistent signatures are recorded in the ledger, the epoch is consolidated.

by Byzantine signers cannot violate safety, and once $f+1$ signatures exist it cannot prevent eventual retrieval through at least one correct signer.

Then, the processing of hash-batch $hb_w$ is continued by adding $w$ to the set of signers in the map *hash_to_signers*, which tracks the servers that appended hash-batches for a given hash $h$ (Line 38). Once the signer set reaches $f + 1$, the server extracts the valid elements from the batch into a set $G$ and assigns an epoch number; we call this step *epoch consolidation* (See Figure 6). The server also adds the valid epoch-proofs contained in the batch to *proofs*. Finally, as in the previous algorithms, the epoch is recorded in *history*, and an epoch-proof for this epoch is generated and sent to the collector via $\texttt{S.add\_to\_batch}_v(ep)$.

## 4. Proofs of Correctness

In this section, we present the proofs of correctness for the Setchain algorithms presented in Section 3.

### *4.1. Correctness of Algorithm Vanilla*

Algorithm Vanilla guarantees Property 1.

**Lemma 1.** *Let* $(T, H, h, P) = \texttt{S.get}_v()$ *be an invocation to a correct server* $v$*. Then,* $\forall i \in \{1, \ldots, h\}, H[i] \subseteq T$*.*

*Proof.* We prove that in a correct server $v$ it holds that $\forall i \in [1, \mathit{epoch}], \mathit{history}[i] \subseteq \mathit{the\_set}$ at all times.

Initially, we have that $\mathit{epoch} = 0$, since Line 14 was never executed. Hence, Line 15 was not executed either, *history* is empty, and the claim trivially holds.

Let us assume now that $\mathit{epoch} > 0$ and consider some $i \in [1, \mathit{epoch}]$. Since *epoch* is only modified in Line 14 by increments of 1, then, when it became equal to $i$, all the elements $G$ that were later added to *history* in epoch $i$ (in Line 15) were guaranteed to be in *the_set* before (in Line 13), and are never removed from *the_set*. □

Algorithm Vanilla guarantees Property 2.

**Lemma 2.** *Let* $\texttt{S.add}_v(e)$ *be an operation invoked on a correct server* $v$*, and* $e$ *a valid element. Then, eventually all invocations* $(T, H, h, P) = \texttt{S.get}_v()$ *satisfy* $e \in T$*.*

*Proof.* The execution of $\texttt{S.add}_v(e)$ by the correct server $v$ will add $e$ to *the_set* (Line 4) if it is not already present. Since elements are never removed from *the_set*, element $e$ will eventually appear in *the_set* returned in all future $\texttt{S.get}_v()$ invocations which return (*the_set*, *history*, *epoch*, *proofs*). □

Algorithm Vanilla guarantees Property 3.

**Lemma 3.** *Let* $v$ *and* $w$ *be two correct servers, let* $e$ *be a valid element, and let* $(T, H, h, P) = \texttt{S.get}_v()$*. If* $e \in T$*, then eventually all invocations* $(T', H', h', P') = \texttt{S.get}_w()$ *satisfy that* $e \in T'$*.*

*Proof.* In Algorithm Vanilla, an element is added by server $v$ to *the_set* in Lines 4 and 13 only.

First, let us consider the case where $e$ is added by server $v$ to *the_set* in Line 4 when a client invoked $\texttt{S.add}_v(e)$. Then, $v$ invokes $\texttt{L.append}_v(e)$ in Line 5. From Property 9, a block $B$ containing $e$ is eventually notified to all correct Setchain servers, including $w$.

On the other hand, if $e$ is added by server $v$ to *the_set* at Line 13 for the first time, then a block $B$ containing $e$ was notified to $v$ by the ledger L. From Property 10, this block must have been notified to all correct Setchain servers, including $w$.

In both cases, when processing block $B$, server $w$ adds the valid element $e$ to *the_set* (Line 13) if it is not already present, and is never removed from *the_set*. Hence, all future invocations of $\texttt{S.get}_w()$ return tuples $(T', H', h', P')$ such that $e \in T'$. □

Algorithm Vanilla guarantees Property 4.

**Lemma 4.** *Let $v$ be a correct server, let $e$ be a valid element and let $(T, H, h, P) = \texttt{S.get}_v()$. If $e \in T$, then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $e \in H'$.*

*Proof.* The element $e$ is added by server $v$ to *the_set* in either Line 4 or 13. First, let us consider the case where $e$ is added by $v$ to *the_set* in Line 4 when a client invoked $\texttt{S.add}_v(e)$. Then, $v$ invokes $\texttt{L.append}_v(e)$ in Line 5. From Property 9, eventually a block $B$ containing $e$ is notified to $v$.

Otherwise, if $e$ is added by server $v$ to *the_set* in Line 13 for the first time, then it was because a block $B$ containing $e$ was notified to $v$ by ledger L.

Then, in both cases, $v$ adds the batch of valid elements $G$ from the block $B$ to *history* at Line 15, with $e \in G$. After that, all future invocations of $\texttt{S.get}_w()$ return tuples $(T', H', h', P')$ that satisfy $e \in H'$. □

Algorithm Vanilla guarantees Property 5.

**Lemma 5.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and let $i, i' \in \{1, \ldots, h\}$ with $i \neq i'$. Then, $H[i] \cap H[i'] = \emptyset$.*

*Proof.* By way of contradiction, let us assume that for some valid element $e$ it holds that $e \in \mathit{history}[i]$ and $e \in \mathit{history}[i']$. Without loss of generality, let us assume that $1 \leq i < i' \leq \mathit{epoch}$.

Observe that the elements of an epoch are inserted into *history* in Line 15, and they are added in increasing epoch order (variable *epoch* is increased every time an epoch is created (Line 14) and never decreased). Then, when $history[i']$ is mapped to a set $G$ with $e \in G$ in Line 15, it already holds that $e \in history[i]$. But, from Line 12, $G$ does not contain elements already in *history*. Hence we have a contradiction. □

Algorithm Vanilla guarantees Property 6.

**Lemma 6.** *Let $v, w$ be correct servers, let $(T, H, h, P) = \texttt{S.get}_v()$ and $(T', H', h', P') = \texttt{S.get}_w()$, and let $i \in \{1, \ldots, \min(h, h')\}$. Then, $H[i] = H'[i]$.*

*Proof.* The proof proceeds by induction on the epoch number $n$. For simplicity we assume that $H[0] = \emptyset = H'[0]$, which is the base case $n = 0$ of the induction. Then, we show for $n > 0$ that $H[n] = H'[n]$, assuming that $\forall i < n, H[i] = H'[i]$.

First, we show that $H[n] \subseteq H'[n]$. (The proof that $H'[n] \subseteq H[n]$ is analogous.) Let $e \in H[n]$. Then $e$ was added by $v$ to $H[n]$ at Line 15 when a ledger block $B$ containing $e$ was processed by $v$. It is easy to see that $B$ is the $n$-th block received by $v$. Therefore, from Property 10, $w$ also receives $B$ as the $n$-th block. The fact that $v$ added element $e$ to $H[n]$ implies that (1) $e$ is a valid element and (2) $e \notin H[0..n-1]$. By inductive hypothesis and (2), we conclude that $e$ is not in $H'[0..n-1]$. Since $e$ is a valid element in the $n$-th block received by $w$ that is not in $H[0..n-1]$, $e$ is added to $H'[n]$. □

Algorithm Vanilla guarantees Property 7.

**Lemma 7.** *Let $v$ be a correct server, $e$ be a valid element, $(T, H, h, P) = \texttt{S.get}_v()$, and $e \in T$. Then there was an operation $\texttt{S.add}_w(e)$ invoked in the past in some server $w$.*

*Proof.* Elements are added to *the_set* in Lines 4 and 13.

First, let us consider the case where $e$ was added by server $v$ to *the_set* in Line 4. Then, it was added when operation $\texttt{S.add}_v(e)$ was being processed.

On the other hand, if $e$ was added by server $v$ to *the_set* in Line 13, then a ledger block $B$ containing $e$ was notified to $v$ by the ledger L. From Property 11, some server $w$ invoked $\texttt{L.append}_w(e)$. Recall that, as mentioned in Section 2.1, we assume that a server cannot create a valid element by itself and that clients and servers do not collude. So, a server $w$ cannot append a valid element $e$ with $\texttt{L.append}_w(e)$ without a client invocation $\texttt{S.add}_w(e)$. □

Algorithm Vanilla guarantees Property 8.

**Lemma 8.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and $i \in \{1, \ldots, h\}$. Then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $P'$ contains at least $f + 1$ epoch-proofs of $H[i]$.*

*Proof.* Let $E = \mathit{history}[i]$ be an epoch with epoch number $i$, whose valid and new elements as set $G$ were added to *history* by $v$ at Line 15. Once this happens, the hash of $G$ is computed and signed by $v$ as $p_v$. Then, an epoch-proof consisting of $\langle i, p_v, v \rangle$ is appended to the ledger by $v$ in Line 17.

By Property 10, all correct Setchain servers will eventually receive enough ledger blocks to reach the creation of epoch $i$. From Line 6, all correct Setchain servers agree on the content of the $i$-th epoch. Therefore, every correct Setchain server $w$ maps $G$ to $\mathit{history}_w[i]$, generates an epoch-proof $\langle i, p_w, w \rangle$ for the i-th epoch, and appends it to the ledger.

Then, from Property 9, ledger blocks containing the epoch-proofs of $\mathit{history}[i]$ will be notified to all correct Setchain servers, including server $v$. Server $v$ will add these epoch-proofs to *proofs* in Line 11. Since we assume a system with $n$ servers, where at most $f < n/2$ are not correct, at least $f + 1$ epoch-proofs for the i-th epoch will be appended to the ledger. Hence, eventually, in all invocations $(T', H', h', P') = \texttt{S.get}_v()$, it will hold that $P'$ will contain at least $f + 1$ epoch-proofs of $\mathit{history}[i]$. □

The combination of all the previous lemmas shows that Algorithm Vanilla implements a Setchain with epoch-proofs.

### *4.2. Correctness of Algorithm Compresschain*

This section follows the same pattern as the previous section, proving Properties 1 to 8 to conclude that Algorithm Compresschain implements a Setchain with epoch-proofs.

Algorithm Compresschain guarantees Property 1.

**Lemma 9.** *Let $(T, H, h, P) = \texttt{S.get}_v()$ be an invocation to a correct server $v$. Then, $\forall i \in \{1, \ldots, h\}, H[i] \subseteq T$.*

*Proof.* We prove that in a correct server $v$ it holds that $\forall i \in [1, \mathit{epoch}], \mathit{history}[i] \subseteq \mathit{the\_set}$ at all times.

Initially, we have that $\mathit{epoch} = 0$, since Line 26 was never executed. Hence, Line 27 was not executed either, *history* is empty, and the claim trivially holds initially.

Let us assume now that $epoch > 0$ and consider some $i \in [1, epoch]$. Since *epoch* is only modified in Line 26 by increments of 1, then, when it became equal to $i$, all the elements $G$ that were later added to *history* in epoch $i$ (in Line 27) were guaranteed to be in *the_set* before (in Line 25), and are never removed from *the_set*. □

Algorithm Compresschain guarantees Property 2.

**Lemma 10.** *Let* $\texttt{S.add}_v(e)$ *be an operation invoked on a correct server* $v$*, and* $e$ *a valid element. Then, eventually all invocations* $(T, H, h, P) = \texttt{S.get}_v()$ *satisfy* $e \in T$*.*

*Proof.* The execution of $\texttt{S.add}_v(e)$ by the correct server $v$ will add $e$ to *the_set* (Line 4) if not already present, which will eventually be returned in all future $\texttt{S.get}_v()$ invocations which return (*the_set*, *history*, *epoch*, *proofs*). □

Algorithm Compresschain guarantees Property 3.

**Lemma 11.** *Let* $v$ *and* $w$ *be two correct servers, let* $e$ *be a valid element, and let* $(T, H, h, P) = \texttt{S.get}_v()$*. If* $e \in T$*, then eventually all invocations* $(T', H', h', P') = \texttt{S.get}_w()$ *satisfy that* $e \in T'$*.*

*Proof.* In Algorithm Compresschain, an element is added by server $v$ to *the_set* in Lines 4 and 25 only.

First, let us consider the case where $e$ is added by server $v$ to *the_set* in Line 4 when a client invoked $\texttt{S.add}_v(e)$. Then, $v$ adds $e$ to *batch* by invoking $\texttt{S.add\_to\_batch}_v(e)$ (Line 5). Eventually, after $e$ is added to the *batch*, a notification $\texttt{isReady}(batch)$ is signaled. Therefore, eventually $v$ compresses *batch* with element $e$ and appends the compressed batch *cb* to the ledger L (Line 15). Note that, elements in *batch* are removed only after the compressed version of *batch* is added to the ledger L (line 16). From Property 9, eventually a block $B$ containing the compressed batch *cb* is notified to all correct Setchain servers, including $w$.

On the other hand, if $e$ is added by server $v$ to *the_set* at Line 25 for the first time, then a block $B$ containing a compressed batch *cb* with $e \in cb$ was notified to $v$ by the ledger L. From Property 10, this block must have been notified to all correct Setchain servers, including $w$.

In both cases, when processing *cb* (with $e \in cb$), server $w$ adds the valid element $e$ to *the_set* (Line 25) if it is not already present and is never removed from *the_set*. Hence, all future invocations of $\texttt{S.get}_w()$ return tuples $(T', H', h', P')$ with $e \in T$. □

Algorithm Compresschain guarantees Property 4.

**Lemma 12.** *Let $v$ be a correct server, let $e$ be a valid element and let $(T, H, h, P) = \texttt{S.get}_v()$. If $e \in T$, then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $e \in H'$.*

*Proof.* The element $e$ was added to *the_set* of server $v$ in Algorithm Compresschain in either Line 4 or Line 25.

First, let us consider the case where $e$ is added by server $v$ to *the_set* in Line 4 when a client invoked $\texttt{S.add}_v(e)$. Then, $v$ adds $e$ to *batch* by invoking $\texttt{S.add\_to\_batch}_v(e)$ (Line 5). Eventually, after $e$ is added to the *batch*, a notification $\texttt{isReady}(batch)$ is signaled. Therefore, eventually $v$ compresses *batch* with element $e$ and appends the compressed batch *cb* to the ledger `L` (Line 15). Note that, elements in *batch* are removed only after the compressed version of *batch* is added to the ledger `L` (Line 16). From Property 9, eventually a block $B$ containing the compressed batch *cb* is notified to $v$.

On the other hand, if $e$ is added to *the_set* at Line 25 for the first time, it was because a block $B$ containing the compressed batch *cb* was notified to $v$ by the ledger `L` and $e \in cb$.

Then, in either case, after receiving *cb* with $e \in cb$ in block $B$, $v$ decompresses *cb* and adds the set $G$ of its valid elements to *history* (if not there already; Line 27). Since $e$ is valid, after this, it holds that $e \in history$. Then, eventually, all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $e \in H'$. □

Algorithm Compresschain guarantees Property 5.

**Lemma 13.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and let $i, i' \in \{1, \ldots, h\}$ with $i \neq i'$. Then, $H[i] \cap H[i'] = \emptyset$.*

*Proof.* By way of contradiction, let us assume that for some valid element $e$ it holds that $e \in history[i]$ and $e \in history[i']$. Without loss of generality, let us assume that $1 \leq i < i' \leq epoch$.

Observe that the elements of an epoch are inserted into *history* in Line 27 and they do it in increasing epoch order. Then, when a set $G$ with $e \in G$ is mapped to $history[i']$ in Line 27, it already holds that $e \in history[i]$. But, from Line 24, $G$ cannot contain elements already in *history*. Hence we have a contradiction. □

Algorithm Compresschain guarantees Property 6.

**Lemma 14.** *Let $v, w$ be correct servers, let $(T, H, h, P) = \texttt{S.get}_v()$ and $(T', H', h', P') = \texttt{S.get}_w()$, and let $i \in \{1, \ldots, \min(h, h')\}$. Then $H[i] = H'[i]$.*

*Proof.* The proof proceeds by induction on the epoch number $n$. For simplicity we assume that $H[0] = \emptyset = H'[0]$, which is the base case $n = 0$ of the induction. Then, we show for $n > 0$ that $H[n] = H'[n]$, assuming that $\forall i < n, H[i] = H'[i]$.

First, we show that $H[n] \subseteq H'[n]$. (The proof that $H'[n] \subseteq H[n]$ is analogous.) Let $e \in H[n]$. Then $e$ was added by $v$ to $H[n]$ at Line 27 when a ledger block $B$ containing the compressed batch $cb$ was processed by $v$ and $e \in cb$. From Property 10, we know that $w$ receives the same set of blocks in the same order as $v$. So, it receives $B$ and processes $cb$ in the same order as $v$ and $e \in cb$ The fact that $v$ added element $e$ to $H[n]$ implies that (1) $e$ is a valid element and (2) $e \notin H[0..n-1]$. By inductive hypothesis and (2), we conclude that $e$ is not in $H'[0..n-1]$. Since $e$ is a valid element that is not in $H[0..n-1]$, $e$ is added to $H'[n]$. □

Algorithm Compresschain guarantees Property 7.

**Lemma 15.** *Let $v$ be a correct server, $e$ be a valid element, $(T, H, h, P) = \texttt{S.get}_v()$, and $e \in T$. Then there was an operation $\texttt{S.add}_w(e)$ invoked in the past in some server $w$.*

*Proof.* Elements are added to *the_set* in the Lines 4 and 25.

First, let us consider the case where $e$ was added by server $v$ to *the_set* in Line 4. Then, it was added when operation $\texttt{S.add}_v(e)$ was being processed.

On the other hand, if $e$ was added by server $v$ to *the_set* in Line 25, then a ledger block $B$ containing a compressed batch $cb$ was notified to $v$ by the ledger L, and $e \in cb$. From Property 11, some server $w$ invoked $\texttt{L.append}_w(cb)$.

Recall that, as mentioned in Section 2.1, we assume that a server cannot create a valid element by itself, and clients and servers do not collude. So, a server $w$ cannot append a valid element $e$ as a part of the compressed batch $cb$ with $\texttt{L.append}_w(cb)$ without a client invocation $\texttt{S.add}_w(e)$. □

Algorithm Compresschain guarantees Property 8.

**Lemma 16.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and $i \in \{1, \ldots, h\}$. Then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $P'$ contains at least $f + 1$ epoch-proofs of $H[i]$.*

*Proof.* Let *history*[$i$] be an epoch with epoch number $i$ whose valid and new elements as set $G$ were added to *history* by $v$ at Line 27. Once this happens, the hash of $G$ is computed and signed by $v$ as $p_v$. Then, an epoch-proof consisting of $\langle i, p_v, v\rangle$ is added to the *batch* by $v$ in Line 29. Eventually the *batch* will be ready, $v$ will compressed *batch* and append the compressed batch $cb'$ to the ledger `L` (line 15).

From Property 10, ledger `L` notifies the same set of blocks in the same order to all servers. Eventually, the ledger block $B$ with the compressed batch $cb$ was notified to each correct server $w$ as it happened to $v$. Then, server $w$ maps valid elements of $cb$ as $G$ to *history*[$i$], generates an epoch-proof $\langle i, p_w, w\rangle$ for the i-th epoch, and appends it to its *batch*. Eventually, this batch will be ready and compressed, and the compressed batch $cb''$ will be appended to the ledger `L`.

Then, from Property 9, ledger blocks containing the compressed batches with the epoch-proofs of the i-th epoch will be notified to all correct Setchain servers, including server $v$. Server $v$ will add these epoch-proofs to *proofs* in Line 23. Since we assume a system with $n$ servers, where at most $f < n/2$ are not correct, at least $f + 1$ epoch-proofs for *history*[$i$] will be appended by correct servers to the ledger as a part of compressed batches. Hence, eventually, in all invocations $(T', H', h', P') = \texttt{S.get}_v()$, it will hold that $P'$ will contain at least $f + 1$ epoch-proofs of *history*[$i$]. □

### *4.3. Correctness of Algorithm Hashchain*

Before writing the proofs for the properties of Setchain, we need some lemmas for *Epoch Consolidation* in Hashchain. Lemma 17 states that when a correct server generates a hash batch and appends it to the ledger, that server is available to share the original batch to other servers when requested.

**Lemma 17.** *Let $v$ be a correct server and let $hb$ be a valid hash batch appended by $v$ to the ledger $L$ through* `L.append`$(hb)$*, where $hb = \langle hs, sg, v\rangle$. Then $v$ is available to share the original batch batch_original corresponding to the hash $hs$ when requested by other servers $w$.*

*Proof.* Two places where a hash batch is appended to the ledger `L` in Algorithm Hashchain are at Lines 19 and 33.

When a correct server $v$ appends a hash batch $hb$ to the ledger `L` at Line 19, it stores the hash $hs$ and the corresponding *batch_original* in local, by calling the `Register_batch`$(h, batch)$ function (Line 16).

When a correct server $v$ appends a hash batch $hb$ to the ledger `L` at Line 33, it has already either retrieved the $batch\_original$ from its local storage (Line 25) or has requested the original batch using the hash $hs$ found in $hb$ (Line 27). Once the batch is retrieved, $v$ proceeds to append the hash batch $hb$ to the ledger `L`, only after verifying (Line 28) that the $batch\_original$ is not empty and the hash of $batch\_original$ matches the hash $hs$ found in $hb$. Also, before appending, it stores the hash $hs$ and the corresponding $batch\_original$ in local, by calling the `Register_batch`$(h, batch)$ function (Line 30).

Hence, in both cases, $v$ will be available to share the original batch, $batch\_original$, corresponding to the hash $hs$ when requested by other servers $w$. □

**Lemma 18.** *Let $v$ be a correct server and let $hb$ be a valid hash batch appended by $v$ to the ledger $L$ through `L.append`$(hb)$, where $hb = \langle hs, sg, v\rangle$. Then eventually $|hash\_to\_signers[hs]| \geq f + 1$ in each correct server $w$.*

*Proof.* When hash batch $hb$ is appended to the ledger `L` by the correct server $v$, by Property 9 a block $B$ containing $hb$ will eventually be notified to all correct Setchain servers.

Each correct Setchain server $w$, when processing $hb$ as a part of the received block $B$, requests the original batch of elements from $v$ after validating the signature $sg$. By previous lemma, $v$ shares the original batch. After receiving the original batch $batch\_original$, $w$ verifies the hash $hs$ as `Hash`$(batch\_original) = hs$. Since the hash is valid ($v$ is correct), then $w$ generates a signature $sg'$ of the hash $hs$. Then, it generates a new hash batch $hb' = \langle hs, sg', w\rangle$ and appends $hb'$ to the ledger. Server $w$ also adds the identity $v$ to $hash\_to\_signers[hs]$.

Again, by Property 9, all the correct Setchain servers will eventually be notified of blocks from the ledger with hash batches for $hs$ from all correct servers. When the hash batch of a correct server is processed, the server is added to $hash\_to\_signers[hs]$.

Since we assume a system with at least $f + 1$ correct servers, eventually $|hash\_to\_signers[hs]| \geq f + 1$. □

Algorithm Hashchain guarantees Property 1.

**Lemma 19.** *Let $(T, H, h, P) =$ `S.get`$_v()$ be an invocation to a correct server $v$. Then, $\forall i \in \{1, \ldots, h\}, H[i] \subseteq T$.*

*Proof.* We prove that in a correct server $v$ it holds that $\forall i \in [1, epoch]$, $history[i] \subseteq the\_set$ at all times.

Initially, we have that $epoch = 0$, since Line 40 was never executed. Hence, Line 42 was not executed either, *history* is empty, and the claim trivially holds.

Let us assume now that $epoch > 0$ and consider some $i \in [1, epoch]$. Since *epoch* is only modified in Line 40 by increments of 1, when it became equal to $i$, all the elements $G$ that were later (in Line 42) added to *history* in epoch $i$ were guaranteed to be in *the_set* before (in Line 37). □

Algorithm Hashchain guarantees Property 2.

**Lemma 20.** *Let* $\texttt{S.add}_v(e)$ *be an operation invoked on a correct server* $v$*, and* $e$ *a valid element. Then, eventually all invocations* $(T, H, h, P) = \texttt{S.get}_v()$ *satisfy* $e \in T$*.*

*Proof.* The execution of $\texttt{S.add}_v(e)$ by the correct server $v$ will add $e$ to *the_set* (Line 4) if not already present. Since the elements are never removed from *the_set*, $e$ will eventually be returned in all future $\texttt{S.get}_v()$ invocations which return (*the_set*, *history*, *epoch*, *proofs*). □

Algorithm Hashchain guarantees Property 3.

**Lemma 21.** *Let* $v$ *and* $w$ *be two correct servers, let* $e$ *be a valid element, and let* $(T, H, h, P) = \texttt{S.get}_v()$*. If* $e \in T$*, then eventually all invocations* $(T', H', h', P') = \texttt{S.get}_w()$ *satisfy that* $e \in T'$*.*

*Proof.* Two places where an element is added to *the_set* in Hashchain are in Lines 4 and 37.

Let us consider the case where in server $v$, $e$ is added to *the_set* in Line 4 when a client invoked $\texttt{S.add}_v(e)$. Then $v$ adds $e$ to *batch* by invoking $\texttt{S.add\_to\_batch}_v(e)$ (Line 5). Once a batch is ready, $v$ invokes $\texttt{L.append}(hb)$, which appends a hash batch $hb$ to the ledger L, where $hb = \langle hs, sg, v \rangle$, $hs = \texttt{Hash}(batch)$, and $e \in batch$.

On the other hand, if $e$ is added to *the_set* at Line 37 for the first time in server $v$, then a block $B$ containing $hb$ was notified to $v$ by the ledger, and the original batch was recovered and is valid. Since the hash is valid, $v$ signs it and generates a new hash batch as $hb = \langle hs, sg, v \rangle$ and appends $hb$ to the ledger L and $e \in batch\_original$.

In either case, according to Property 9, eventually a block $B$ containing $hb$ is notified to all correct Setchain servers, including $w$. When $w$ encounters the hash batch with the hash $hs$ as a part of block $B$, eventually $w$ could retrieve the original batch from $v$ as proved in Lemma 17, if it doesn't have it already. After verifying the original batch corresponding to $hb$, $w$ would add the valid element $e$ to *the_set* if it is not already present (Line 37). So, eventually all future invocations of $\texttt{S.get}_w()$ returns (*the_set*, *history*, *epoch*, *proofs*) and $e \in$ *the_set*. □

Algorithm Hashchain guarantees Property 4.

**Lemma 22.** *Let $v$ be a correct server, let $e$ be a valid element and let $(T, H, h, P) = \texttt{S.get}_v()$. If $e \in T$, then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $e \in H'$.*

*Proof.* Two places where an element is added to *the_set* in Hashchain are in Lines 4 and 37.

Let us consider the case where in server $v$, $e$ is added to *the_set* in Line 4 when a client invoked $\texttt{S.add}_v(e)$. Then $v$ adds $e$ to *batch* by invoking $\texttt{S.}add\_to\_batch_v(e)$ (Line 5). Once a batch is ready, $v$ invokes $\texttt{L.append}(hb)$, which appends a hash batch $hb$ to the ledger L, where $hb = \langle hs, sg, v\rangle$, $hs = \texttt{Hash}(batch)$, and $e \in batch$.

On the other hand, if $e$ is added to *the_set* at Line 37 for the first time in server $v$, then a block $B$ containing $hb' = \langle hs, sg', w\rangle$ was notified to $v$ by the ledger, and the original batch was recovered and is valid. Since the hash is valid, then $v$ signs the hash $hs$ and generates a new hash batch for the hash $hs$ as $hb = \langle hs, sg, v\rangle$ and appends $hb$ to the ledger L. Also, it adds $e$ to *the_set*.

According to Lemma 18, eventually $v$ receives at least $f + 1$ signatures for hash $hs$. When $v$ receives the $f + 1$-th signature, the batch consolidates and it is assigned an epoch number (Line 40). Then, the valid elements from the batch, including $e$, are added to *history* as an epoch (Line 42), if not already present. Eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $e \in H'$. □

Algorithm Hashchain guarantees Property 5.

**Lemma 23.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and let $i, i' \in \{1, \ldots, h\}$ with $i \neq i'$. Then, $H[i] \cap H[i'] = \emptyset$.*

*Proof.* By way of contradiction, let us assume that for some valid element $e$ it holds that $e \in history[i]$ and $e \in history[i']$. Without loss of generality, let us assume that $1 \leq i < i' \leq epoch$.

So, for $e$ to be included in the epoch $i'$ as a part of $G$ (Line 42), $G$ must have been processed through Line 36, which allows only elements that are not already present in *history* to pass through. We know that $i < i'$ and *epoch* is only increased (Line 40) and never decreased. So, if epoch $i$ was processed first and, as a result, $e \in history[i]$, then the attempt to add $e$ to $history[i']$ would not clear the conditions in Line 36. So, this contradicts our assumption that $e \in history[i']$. Then, we have $history[i] \cap history[i'] = \emptyset$. □

Algorithm Hashchain guarantees Property 6.

**Lemma 24.** *Let $v, w$ be correct servers, let $(T, H, h, P) = \texttt{S.get}_v()$ and $(T', H', h', P') = \texttt{S.get}_w()$, and let $i \in \{1, \ldots, \min(h, h')\}$. Then $H[i] = H'[i]$.*

*Proof.* The proof proceeds by induction on the epoch number $n$. For simplicity we assume that $H[0] = \emptyset = H'[0]$, which is the base case $n = 0$ of the induction. Then, we show for $n > 0$ that $H[n] = H'[n]$, assuming that $\forall i < n, H[i] = H'[i]$.

First, we show that $H[n] \subseteq H'[n]$. (The proof that $H'[n] \subseteq H[n]$ is analogous.) Let $e \in H[n]$. Then $e$ that was added by $v$ to $H[n]$ at Line 42 when $v$ received the $f + 1$-th signature for the hash $hs$ as $hb = \langle hs, sg, z \rangle$, such that $v.hash\_to\_batch[hs] = batch\_original$, $e \in batch\_original$, $\texttt{Hash}(batch\_original) = hs$, and $|hash\_to\_signers[hs]| = f + 1$. The fact that $v$ added element $e$ to $H[n]$ implies that (1) $e$ is a valid element and (2) $e \notin H[0..n-1]$. Then, by Property 9, $w$ will eventually receive a ledger block containing $hb = \langle hs, sg, z \rangle$ and by Properties 9 and 10, it will be the $f+1$-th signature $w$ receives for hash $hs$. Since $w$ has received $f+1$ signatures of $hs$, at least one of the signers is correct, and by Lemma 17, a correct server $z$ was available to share $batch\_original$ with $w$ when $w$ requested it at Line 27 (if $w$ already doesn't have $batch\_original$).By inductive hypothesis and (2), we conclude that $e$ is not in $H'[0..n-1]$. Since $e$ is a valid element that is not in $H'[0..n-1]$, it is added to $H'[n]$ at Line 42 by $w$. □

Algorithm Hashchain guarantees Property 7.

**Lemma 25.** *Let $v$ be a correct server, $e$ be a valid element, $(T, H, h, P) = \texttt{S.get}_v()$, and $e \in T$. Then, there was an operation $\texttt{S.add}_w(e)$ invoked in the past in some server $w$.*

*Proof.* Elements are added to *the_set* in the Lines 4 and 37. First, let us consider the case where $e$ was added by server $v$ to *the_set* in Line 4. Then, it was added when operation $\texttt{S.add}_v(e)$ was being processed.

On the other hand, if $e$ was added by server $v$ to *the_set* in Line 37, then a ledger block $B$ containing a hash batch $hb$ was notified to $v$ by the ledger L, where $hb = \langle hs, sg, v\rangle$, $hs = \texttt{Hash}(batch)$, and $e \in batch$. From Property 11, some server $w$ invoked $\texttt{L.append}_w(hb)$.

Recall that, as mentioned in Section 2.1, we assume that a server cannot create a valid element by itself and do not collude with clients. So, a server $w$ cannot append a valid element $e$ as a part of the hash batch $hb$ with $\texttt{L.append}_w(cb)$ without a client invocation $\texttt{S.add}_w(e)$. □

Algorithm Hashchain guarantees Property 8.

**Lemma 26.** *Let $v$ be a correct server, $(T, H, h, P) = \texttt{S.get}_v()$, and $i \in \{1, \ldots, h\}$. Then eventually all invocations $(T', H', h', P') = \texttt{S.get}_v()$ satisfy that $P'$ contains at least $f + 1$ epoch-proofs of $H[i]$.*

*Proof.* Let $E = history[i]$ be an epoch with epoch number $i$ added to *history* by $v$ at Line 42. Then $v$ received the $f + 1$-th signature for the hash $hs$ as $hb = \langle hs, sg, z\rangle$, such that $v.hash\_to\_batch[hs] = batch\_original$, $e \in batch\_original$, $\texttt{Hash}(batch\_original) = hs$, and $|hash\_to\_signers[hs]| = f + 1$. From Properties 9 and 10, ledger L notifies the same set of blocks in the same order to all servers and the transactions inside a ledger block are ordered as well. So, all correct Setchain servers $w$, eventually receive enough ledger blocks to reach the consolidation of epoch $i$.

Once the epoch is consolidated, each correct server $w$ generates the proof for the epoch $E$ as $\langle i, p_w, w\rangle$ and append it to *batch*. Eventually, the batch is ready, $w$ hashes *batch*, signs it and appends it to the ledger as $ohb = \langle ohs, osg, w\rangle$. By Property 9, eventually $v$ receives a block $B$ containing $ohb$ and by Lemma 17, $v$ can retrieve the *batch* from $w$, which contains the epoch-proof $\langle i, p_w, w\rangle$. Then, $v$ adds the proof of epoch $E$ to $proofs$ in Line 35. Since we assume a system with $n$ servers, where at most $f < n/2$ are not correct, eventually $v$ will receive and add to $proofs$ at least $f+1$ epoch-proofs for epoch $E$, and $proofs$ will be appended to the ledger as a part of the hash batches. Hence, eventually, in all invocations $(T', H', h', P') = \texttt{S.get}_v()$, it will hold that $P'$ will contain at least $f + 1$ epoch-proofs of $E$. □

## 5. Analytical Performance Study

In this section, we present a brief analysis of the stationary throughput that each of these algorithms can achieve. Let us assume in this analysis that all the $n$ system servers are correct. Then, each epoch will have $n$ epoch-proofs appended to the ledger L in the case of Algorithm Vanilla, and sent to the collector to be added to the *batch* in the case of Compresschain and Hashchain. Let us assume epoch-proofs have length $l_p$, the elements added by the clients have length $l_e$, and the ledger L has blocks of capacity $C$. With Algorithm Vanilla the valid elements in each ledger block form an epoch. Then, in the steady state, each block will contain $n$ epoch-proofs and up to $(C - n \cdot l_p)/l_e$ elements[3]. If ledger L creates blocks at a rate $R$ (in blocks/second), Vanilla can reach a throughput of $T_v = R(C - n \cdot l_p)/l_e$ elements/second. Let us now consider Compresschain with collector size $c$. In the steady state, for each epoch $n$ epoch-proofs are generated. This means that, on average, there are $n$ epoch-proofs in each batch. Let us assume the compression algorithm used has a compression ratio $r$. Then, the length in the ledger of an epoch created from a full collector is $\ell = ((c - n) \cdot l_e + n \cdot l_p)/r$. Then, in the steady state, each ledger block will contain up to $(c - n) \cdot C/\ell$ valid elements, and Compresschain can reach a throughput of $T_c = \frac{R \cdot (c-n) \cdot C}{\ell} = \frac{R \cdot (c-n) \cdot C}{((c-n) \cdot l_e + n \cdot l_p)/r}$ elements/second. Finally, let us consider Hashchain with collector size $c$, and let $l_h$ be the length of a hash-batch. Let us assume that the bottleneck of Algorithm Hashchain is appending to the ledger L. Observe that $n$ hash-batches are appended for each epoch consolidated. Then, in the steady state, Hashchain can reach a throughput of $T_h = R \cdot (c - n) \cdot C/(n \cdot l_h)$ elements/second.

## 6. Implementation and Evaluation Environment

We implement the three proposed algorithms in Golang, using CometBFT as the underlying block-based ledger. We first outline key implementation aspects, and then describe the experimental environment used to evaluate these implementations.

[3]For simplicity we will omit floors and ceilings in this analysis.

### *6.1. CometBFT*

CometBFT [15] (formerly known as Tendermint) is a Byzantine-tolerant state machine replication engine. It is a blockchain middleware that supports the replication of arbitrary applications implemented in any programming language. Two fundamental components of CometBFT are a blockchain *consensus engine* and a generic *application interface*. The consensus engine is called *TendermintCore* [11] and ensures that every validator (server) agrees on the same sequence of blocks with the same set of transactions in the same order. CometBFT (TendermintCore) preserves safety as long as strictly less than one third of the validators' voting power is Byzantine; in the common equal-voting-power setting this corresponds to the standard bound $n \geq 3f + 1$, i.e., $f = \lfloor (n-1)/3 \rfloor$. The application interface called *Application BlockChain Interface (ABCI)*, bridges the consensus engine and the application. We write the code for the Setchain algorithms in the ABCI section of the ledger.

### *6.2. Mapping the Block-based Ledger to CometBFT.*

The block-based ledger has two endpoints: an `Append` function and a `NewBlock` notification. Here we define how these endpoints map to CometBFT.

- `Append`: In our algorithms, when a client appends a transaction, it uses the function `BroadcastTxAsync` to send the transaction to a CometBFT ledger server. This function sends the transaction to the server without waiting for a reply. The server stores the transaction in the mempool and checks if the transaction is valid before sharing it with the other servers using a gossip protocol.

- `NewBlock`: CometBFT's application interface ABCI has a function `FinalizeBlock`. When the CometBFT validators agree upon the proposed block and finalize the order of transactions, the block is sent to all the CometBFT servers for the application layer to process the block. This allows the application to generate additional data (e.g., events, updates to the application's state, etc.) after the block has been finalized. So, the process done in the Setchain algorithms upon `NewBlock` notification is done as a part of CometBFT's `FinalizeBlock` application logic.

Table 3: Parameters for Setchain evaluation

| Name | Description | Values |
|---|---|---|
| *sending_rate* | Adding rate (el/s) | 10000, 5000, 1000, 500 |
| *collector_limit* | Collector size (el) | 100, 500 |
| *server_count* | Number of servers | 4, 7, 10 |
| *network_delay* | Delay increase (ms) | 0, 30, 100 |

### *6.3. Performance Evaluation Platform*

We conducted a performance evaluation of the Setchain using a cluster. Each machine in the cluster has an Intel(R) Xeon(R) E-2186G CPU @ 3.80 GHz with 12 cores, 32 GB RAM, and runs Debian GNU/Linux 11 (bullseye). We use Docker Engine version 20.10.5. The Setchain algorithms are implemented on top of CometBFT v0.38. Each ledger server runs in a Docker container, and each container runs on a separate machine in the cluster. The containers have no CPU or memory limit. Each Docker container contains one client, one collector module, and one CometBFT server (which contains the Setchain functionality in the ABCI).

### *6.4. Experiment Scenarios*

The experimental parameters used in this study are summarized in Table 3. The sending rate reported in Table 3 corresponds to the aggregate rate at which elements are injected by all clients. Each client sends elements at a rate of $sending\_rate/server\_count$, forwarding them to its local server (i.e., the server running in the same Docker container). The parameter $network\_delay$ represents an artificial latency added to all inter-server communication, in order to emulate the impact of moving from a local cluster to a wide-area deployment. CometBFT's mempool plays a key role: it stores unconfirmed ledger transactions after validation and before they are included in blocks. In the default CometBFT configuration, the mempool admits at most 5,000 transactions. Since this limit could become a bottleneck for our Setchain evaluation, we increased the mempool capacity (after preliminary trial-and-error tuning) to 10,000,000 transactions or 2 GB, whichever is reached first.

Unless stated otherwise, each experiment is configured so that clients inject elements into the Setchain for 50 seconds. The experiment terminates once all injected elements have been placed into epochs and all epoch-proofs

have been appended to ledger blocks; we then collect and analyze the logs. Injection and throughput are computed over the Setchain elements sent by clients and are reported in *elements per second (el/s)*. To keep the workload realistic, we use transactions downloaded from Arbitrum [31] as Setchain elements. Hashing uses SHA512 [36], signatures use ed25519 from the EdDSA family [30, 7], and compression uses Brotli [5]. The average Arbitrum transaction size is approximately 438 bytes, with a standard deviation of 753.5. The length of an epoch-proof is 139 bytes. In Compresschain, the average compressed batch size is approximately 16,000 bytes with a standard deviation of 2,100 for a collector limit of 100, and approximately 66,000 bytes with a standard deviation of 15,000 for a collector limit of 500; the corresponding compression ratio thus ranges roughly from 2.5 to 3.5. In Hashchain, the size of a hash-batch is 139 bytes. In CometBFT, blocks are produced roughly every 1.25 seconds (i.e., the block rate is approximately 0.8 blocks/s). Unless otherwise stated, the CometBFT block size is set to 0.5 MB.

## 7. Performance Results

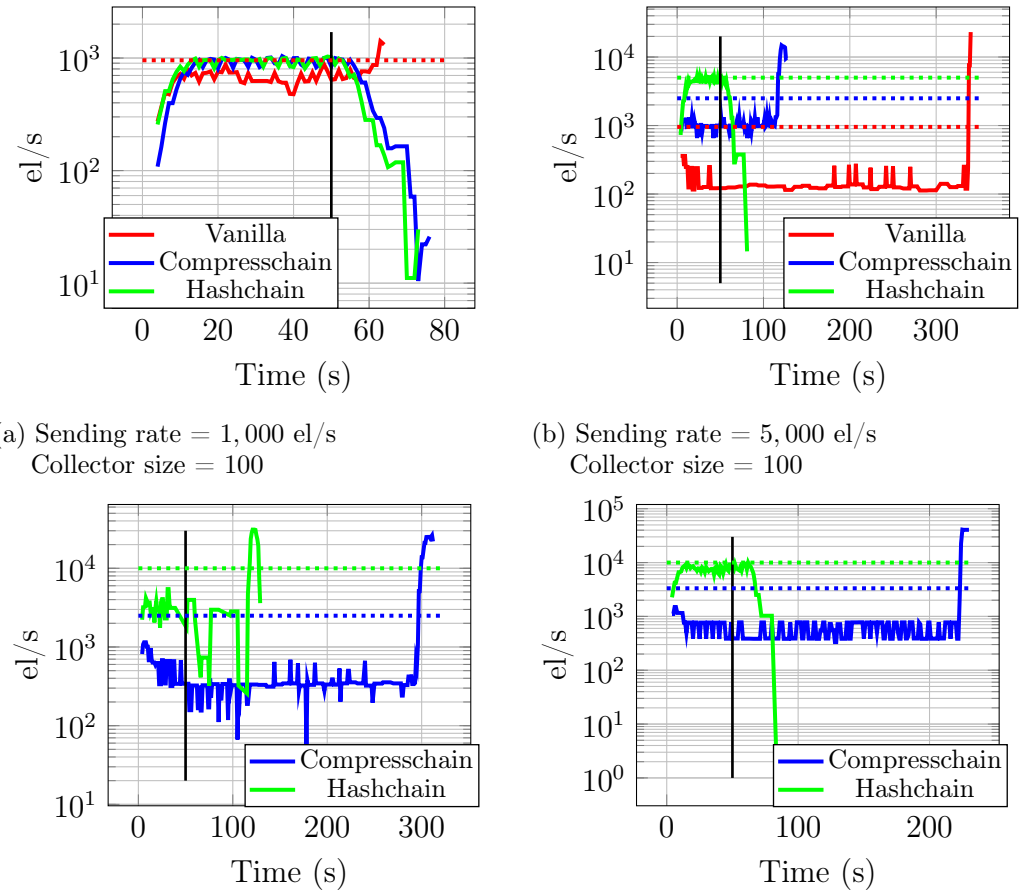
### *7.1. Analysis*

We can estimate the achievable throughput for each algorithm using the analysis of Section 5 with the parameters of the evaluation scenario described in Section 6. These parameters for Algorithm Vanilla are, in the average case, $n = 10$, $C = 500,000$ bytes, $l_e = 438$ bytes, $l_p = 139$ bytes, and $R = 0.8$ block/s. Hence, $T_v \approx 955$ el/s. For Algorithm Compresschain we found that for collector size $c = 100$ the compression ratio is roughly $r = 2.7$. Hence, we have $T_c[c = 100] \approx 2,497$ el/s. For collector size $c = 500$ the compression ratio is roughly $r = 3.5$, and we have $T_c[c = 500] \approx 3,330$ el/s. Finally, for Algorithm Hashchain, using that the hash-batch has length $l_h = 139$ bytes, for collector size $c = 100$ we have $T_h[c = 100] \approx 27,157$ el/s, while for collector size $c = 500$ we have $T_h[c = 500] \approx 147,857$ el/s.

Observe that $T_h[c = 500]/T_v \approx 155$ and $T_h[c = 500]/T_c[c = 500] \approx 44$. Hence, with Hashchain we expect the throughput to increase significantly.

### *7.2. Throughput Comparison*

Fig. 7 reports the throughput over time (in *committed* elements per second, el/s) achieved by the three Setchain algorithms under different sending rates, using 10 servers and no additional network delay. An added element is considered *committed* once the epoch that contains it has accumulated at




(a) Sending rate = 1,000 el/s
Collector size = 100

(b) Sending rate = 5,000 el/s
Collector size = 100

(c) Sending rate = 10,000 el/s
Collector size = 100

(d) Sending rate = 10,000 el/s
Collector size = 500

Figure 7: Throughput over time of the Setchain algorithms for different sending rates. Solid lines plot the rolling average number of elements committed in 9 seconds. The vertical bar marks the time clients add the last element (roughly after 50 s). The dotted horizontal lines show the minimum of the sending rate and the analytical throughput computed in Section 7.1.

least $f + 1$ epoch-proofs in the ledger. For reference, the figure also includes the analytical throughput values derived in Section 7.1.

In Fig. 7a, with a sending rate of 1,000 el/s and collector size 100, all three Setchain variants behave well. Still, the end-of-run peak for Vanilla suggests moderate stress, as many epoch-proofs accumulate near completion. This is

Table 4: Throughput Comparison (upto 50s) for Figure 7

| **Algorithms** | **Figure 7a** | **Figure 7b** | **Figure 7c** | **Figure 7d** |
|---|---|---|---|---|
| Vanilla | 657 el/s | 171 el/s | 100 el/s | 100 el/s |
| Compresschain | 845 el/s | 996 el/s | 571 el/s | 743 el/s |
| Hashchain | 884 el/s | 4,183 el/s | 2,540 el/s | 7,369 el/s |

consistent with the maximum throughput for this configuration computed in Section 7.1, which is 911 el/s and therefore below the 1,000 el/s injection rate.

In Fig. 7b, for a sending rate of 5,000 el/s and collector size 100, both Vanilla and Compresschain require a long time to commit all submitted elements and exhibit a pronounced end peak, again indicating stress. In addition, both achieve throughput below the analytical bound, and even below the throughput observed in Fig. 7a. In contrast, Hashchain sustains the load and completes shortly after the injection phase ends.

Fig. 7c shows the throughput for a sending rate of 10,000 el/s with collector size 100. We omit Vanilla in order to focus the comparison between Compresschain and Hashchain. In this setting, both algorithms show signs of stress, although Compresschain is affected much more than Hashchain. As further illustrated in Fig. 7d, increasing the collector size to 500 alleviates stress for Hashchain, whereas it provides limited benefit for Compresschain.

Finally, Table 4 reports the average throughput up to 50 s for all three algorithms across the experiments shown in Fig. 7.

*7.2.1. Pushing the Hashchain Limits.*

As shown in Fig. 7d, there is a clear gap between the analytically achievable throughput (147,857 el/s; see Section 7.1) and the throughput attained by our Hashchain implementation, mainly because our earlier experiments did not explore sufficiently high sending rates. To identify the maximum achievable throughput in practice, we therefore increased the sending rate. We observed a bottleneck around 20,000 el/s, which persisted even when further increasing the sending rate beyond this point and regardless of the collector size (see Fig. 8). The most plausible explanation is the hash-reversal step, in which servers exchange transaction batches over the network for each batch-hash produced by the collector.

In our current implementation, Setchain servers are responsible for distributing transaction batches. More efficient designs are possible, for example

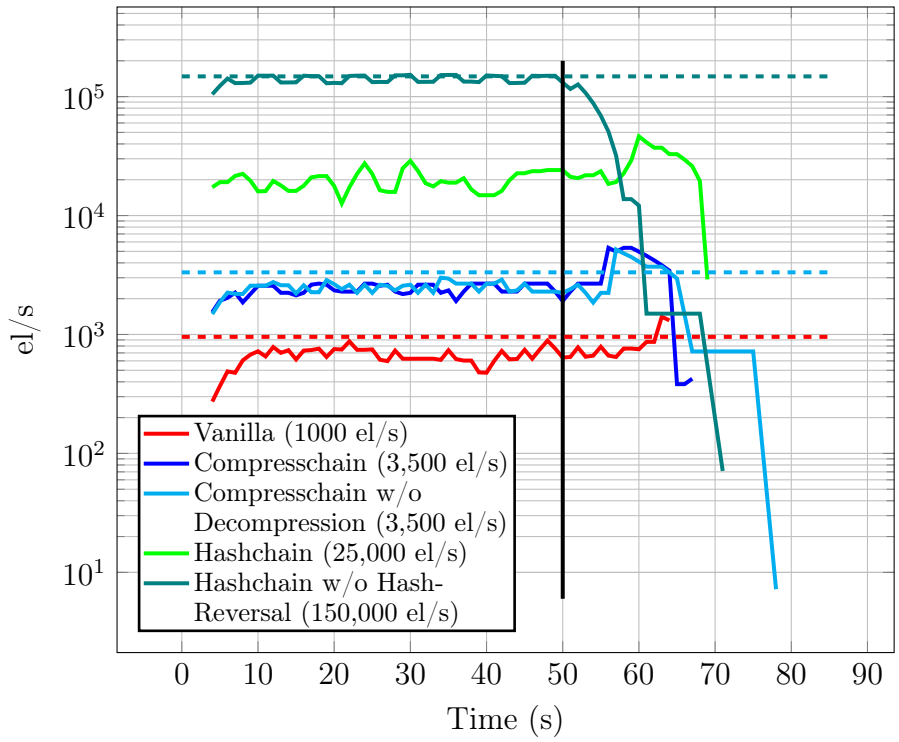



Figure 8: Highest throughput measured in the evaluation platform (with collector size 500 when required). The vertical bar marks the time clients add the last element (roughly after 50 s). Solid lines plot the rolling average number of elements committed in 9 s. The dashed lines show the analytical throughput.

by having only a subset of $2f + 1$ servers sign each batch-hash and epoch, by using optimistic validation of hash-batches (i.e., transferring a batch only upon request), or by adopting alternative batch-sharing protocols. To quantify the cost of hash-reversal, we ran additional experiments in which we removed hash-reversal and hash-batch validation, while assuming that all servers are correct (and thus that all hash-batches are valid).

Fig. 8 reports the highest throughput achieved by Hashchain both with and without hash-reversal. The results indicate that hash-reversal is indeed the dominant bottleneck: with a more efficient hash-reversal mechanism, Hashchain would scale substantially better. Without hash-reversal, Hashchain reaches an average throughput of $133,882$ el/s during the first 50 seconds with a sending rate of $150,000$ el/s, whereas with hash-reversal

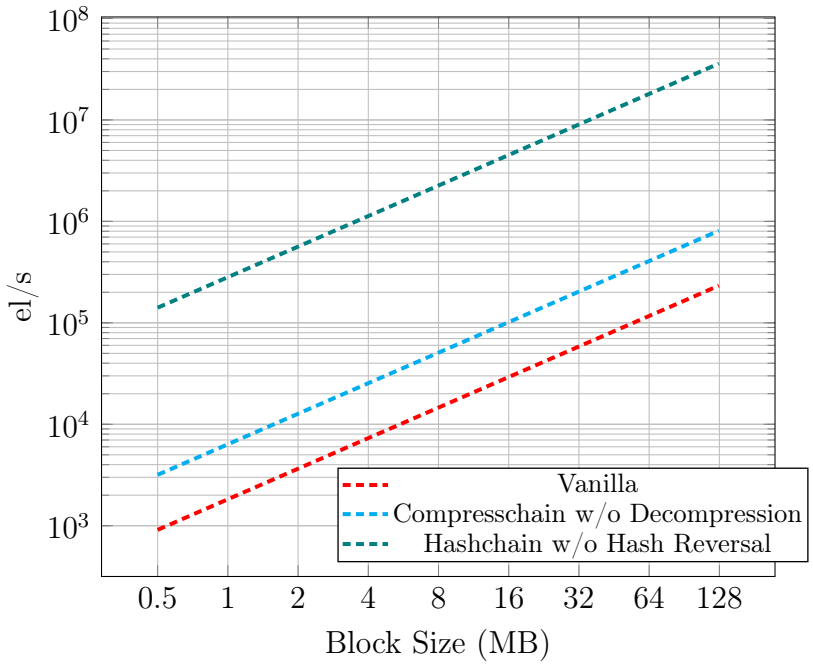
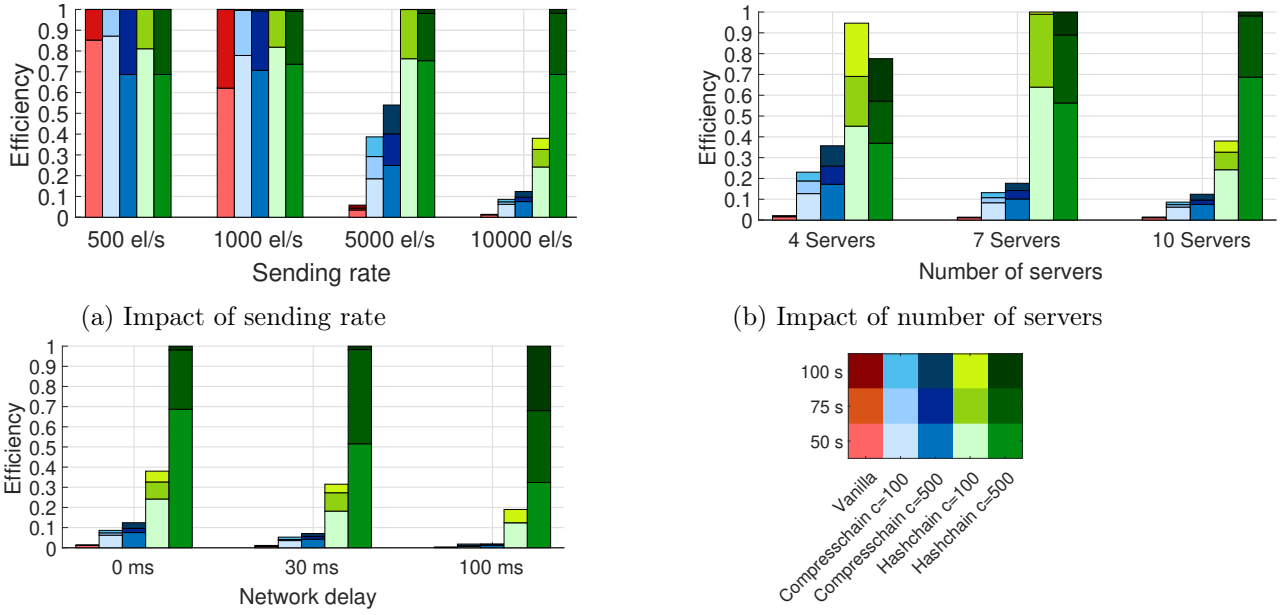



Figure 9: Analytical throughput of the Setchain algorithms for various block sizes with collector size 500.

enabled it reaches only 20,061 el/s during the first 50 seconds with a sending rate of 25,000 el/s. These measurements use a collector size of 500 to remain comparable with the other experiments in this work; increasing the collector size would likely yield even higher throughput when hash-reversal is disabled.

Fig. 8 also reports the highest throughputs achieved by Compresschain and Vanilla, for comparison with Hashchain. For Algorithm Compresschain, we additionally run the implementation with and without decompression and validation, to isolate the impact of these steps. In all cases, the observed throughputs remain well below Hashchain's throughput, even when hash-reversal is enabled.

Finally, the highest throughputs we observe for Vanilla, Compresschain without decompression, and Hashchain without hash-reversal are all close to their respective analytical bounds. Fig. 9 presents the analytical throughput of the three Setchain algorithms under larger CometBFT block sizes, keeping the remaining parameters unchanged (for Algorithm Compresschain and Algorithm Hashchain, we use a collector size of 500). As illustrated in




(a) Impact of sending rate

(b) Impact of number of servers

(c) Impact of network delay

(d) Color codes

Figure 10: Efficiency values observed under different scenarios. The base scenario has 10 servers, a sending rate of 10,000 el/s, and no (0) network delay.

the figure, with the typical 4 MB CometBFT block size, Hashchain reaches a throughput of $10^6$ el/s, and with 128 MB blocks it exceeds 30 million el/s.

### *7.3. Efficiency*

To quantify how stressed an algorithm is, we introduce a metric called *efficiency*. We define efficiency as the ratio between the number of committed elements and the total number of elements added. We report this efficiency after 50, 75, and 100 seconds. Recall that, in every experiment, clients inject elements for 50 seconds. When an algorithm is not stressed, we expect efficiency to be close to 1 at 50 seconds, and equal to 1 by 75 seconds. Unless stated otherwise, we use a baseline configuration with 10 servers, a sending rate of 10,000 el/s, and 0 network delay, and we vary one parameter at a time.

#### *7.3.1. Impact of the Sending Rate.*

Fig. 10a reports efficiency for four sending rates (500, 1,000, 5,000, and 10,000), and for collector sizes 100 and 500 when applicable. The figure

shows that, at the lower rates 500 and $1,000$, all algorithms reach full efficiency within 70 seconds. At $5,000$ and $10,000$, Vanilla exhibits very low efficiency. Compresschain also experiences a marked drop in efficiency, and increasing the collector size from 100 to 500 provides limited improvement. Hashchain shows reduced efficiency only at $10,000$, and this effect is mitigated by increasing the collector size.

*7.3.2. Impact of Number of Servers.*

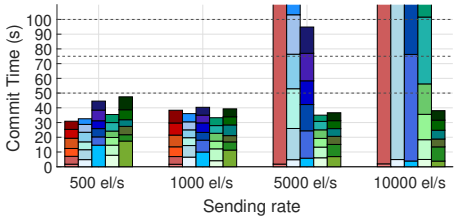
In Fig. 10b, we study the effect of varying the number of servers while keeping the sending rate fixed at $10,000$ el/s. Vanilla consistently exhibits the lowest efficiency, even with 4 servers. Compresschain also achieves low efficiency, with only marginal improvement from increasing the collector size; moreover, its efficiency decreases as the number of servers grows. Hashchain shows low efficiency only with 10 servers and collector size 100. Interestingly, Hashchain also shows slightly imperfect efficiency with 4 servers, which may be explained by having fewer servers available to support the reverse hashing process.

*7.3.3. Impact of Network Delay.*

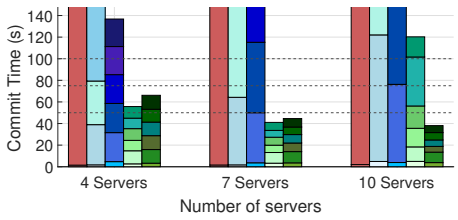
Fig. 10c illustrates how adding artificial delay to all inter-server messages affects efficiency, modeling the transition from a local-area network to a wide-area network. As expected, increasing the network delay reduces efficiency. Nevertheless, even at the largest delay of 100 ms, Hashchain with collector size 500 reaches full efficiency within 100 seconds.

*7.4. Commit Time Comparison*

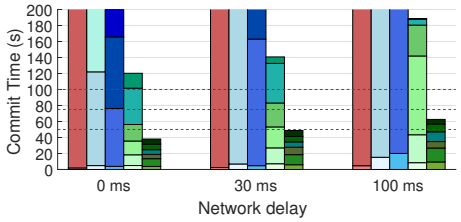
In this section, we explore how elements get committed over time in the same scenarios explored in Section 7.3. For that, we compute the commit time of the first element, followed by the 10%, 20%, 30%, 40%, and 50% of the elements for each algorithm and scenario. We plot this data in Fig. 11 with the $y$ axis truncated for visibility. In Fig. 11a we present results for different sending rates. We observe that Vanilla commits the first element earlier than the other two algorithms. For the low rates (500 and $1,000$) elements are committed at a regular pace. However, in the seven combinations of rate and algorithm that showed low efficiency, the pace is not regular. Fig. 11b shows the impact of the number of servers in the commit time. A higher number of servers increases the commit time for Vanilla (although barely observable in the figure) and Compresschain, possibly because it makes consensus harder



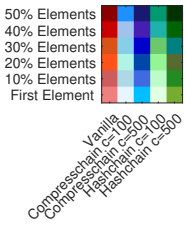

(a) Impact of sending rate (10 servers, 0 network delay).




(b) Impact of number of servers (10,000 el/s, 0 network delay).

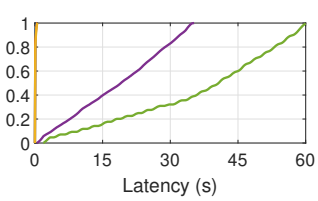



(c) Impact of network delay (10 servers, 10,000 el/s).

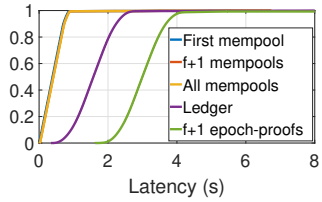



(d) Color codes

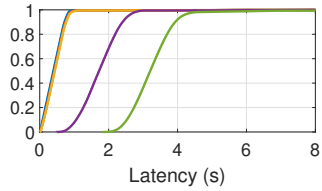
Figure 11: Commit times observed under different scenarios. The base scenario has 10 servers, a sending rate of 10,000 el/s, and no (0) network delay.




(a) Vanilla




(b) Compresschain (collector size = 100)




(c) Hashchain (collector size = 100)

Figure 12: Cumulative distribution function $F(x)$ of the latency experienced by the elements added to the Setchain to reach several stages in their process: (1) reach the mempool, (2) be replicated in $f+1$ server mempools, (3) be replicated in all server mempools, (4) be inserted in a ledger block, and (5) $f+1$ epoch-proofs are present in the ledger. The scenario is with 10 servers, a sending rate of 1,250 el/s, and no network delay.

to reach. Surprisingly, for Hashchain (especially for a collector size of 500) the pattern seems to be the opposite. We believe that a larger number of servers helps with the reverse hashing process of Hashchain. Regarding the

impact of the network delay, shown in Fig. 11c, the conclusion is that it negatively affects the commit times, as expected.

### *7.5. Latency*

When a client sends a valid element to its local Setchain server, that element is eventually submitted to a CometBFT server as part of a ledger transaction. Once validated, the transaction is added to the mempool of the CometBFT server. Subsequently, it is disseminated to other CometBFT servers via a gossip protocol that propagates mempool transactions to peers, who in turn validate and forward them. Eventually, the transaction is included in a ledger block, and Setchain servers append epoch-proofs to the ledger. Once $f + 1$ epoch-proofs for the epoch containing the element have been included in ledger blocks, the element is considered committed. Fig. 12 reports the latency distributions observed for elements added to the Setchain, decomposed into five stages up to commit, for each algorithm. Specifically, it shows the time for an element to reach: (1) the mempool of one CometBFT server, (2) the mempools of $f + 1$ CometBFT servers, (3) the mempools of all CometBFT servers, and (4) the ledger. Finally, it shows (5) the commit latency, defined as the time until $f + 1$ epoch-proofs for the epoch containing the element appear in the ledger. In Fig. 12a, mempools are reached almost immediately because Vanilla forwards elements directly to the CometBFT server. Compresschain and Hashchain, in contrast, wait until the collector is full (or a timeout expires) before submitting the resulting batch as a ledger transaction, which explains the delayed mempool latencies in Fig. 12b and Fig. 12c. Consider now the time to reach the ledger and then to commit (i.e., to obtain $f + 1$ epoch-proofs). With Vanilla, these gaps are on the order of tens of seconds in most cases, whereas for Compresschain and Hashchain they are typically between one and two seconds. In this scenario, both Compresschain and Hashchain achieve commit latency and finality below 4 seconds with probability close to 1.

## 8. Conclusion and Discussions

In this work, we present three real-world Setchain implementation algorithms built on top of a block-based ledger. We formally verify that these algorithms satisfy the Setchain properties. We also implement all three algorithms and carry out an empirical evaluation to assess the effectiveness of each approach across a range of scenarios. Overall, Hashchain consistently

achieve the best results across the evaluated metrics. By leveraging the succinctness of hashing, it improves scalability and is therefore well-suited to large-scale deployments. Compresschain provides improvements over Vanilla, but its gains remain below those obtained with Hashchain.

### *8.1. Extension to a fully functional blockchain*

While the algorithms proposed are meant to implement a Setchain object, in which elements inside an epoch have no order, in their implementation, they impose an order inside each epoch (e.g., to hash the epoch elements consistently). This means that the proposed algorithms can be easily extended to implement a blockchain, similarly as how Hyperledger Fabric or RedBelly do:

- When adding elements to the Setchain and creating epochs, each transaction can be optimistically validated by itself (independently of all other transactions, that is, in parallel), ignoring its semantics.
- After each epoch is consolidated and its transactions ordered, the effect of its transactions can be computed (sequentially) in its actual final position. If a transaction is determined to be invalid it is marked as void.

Observe that extending Setchain to a blockchain can present a trade-off between decentralization and scalability. As transaction execution must be done sequentially in an epoch, large epochs may require large computational resources. To ensure that less powerful servers can maintain synchronization with the blockchain, it may be required to limit epoch sizes, similar to Ethereum's block size limitations [3].

### *8.2. Limitations and Future Work*

To the best of our knowledge, there is currently no blockchain benchmarking tool that allows for a direct comparison between Setchain and state-of-the-art blockchain systems. We considered Diablo [27], a benchmarking suite focused on evaluating decentralized applications (dApps), but it does not suit our needs as Setchain does not currently support dApps.

Another limitation of the current work is that we do not incorporate explicit defenses against transaction-flooding or DDoS-style overload. Our system model allows an unbounded number of clients, including Byzantine

clients, and under aggressive injection, the main expected effect is performance degradation: requests may accumulate in the collector or mempool, increasing latency and reducing efficiency when the offered load exceeds the service capacity. This tendency is already visible in our experiments as the sending rate increases. Designing admission-control, rate-limiting, or other anti-overload mechanisms for Setchain-based systems is an interesting direction for future work.

There are several other potential directions in which future work could evolve. One would be to run the experiments in a more distributed environment to understand the scalability limits and ensure the system can handle real-world demands. This would also include understanding the tradeoffs between system parameters.

While using an underlying block-based ledger service, like Comet-BFT, simplifies the algorithms, we have observed that it may also be a bottleneck. We will explore if replacing this service with something lighter, like a set consensus service [18] or Malachite [45], can increase performance and scalability.

Another interesting work, as highlighted in Section 7.2.1, would be to implement a more efficient hash-reversal technique that could further improve Hashchain's performance. Another promising direction would be to design more efficient APIs for Setchain, and implement specific DeFi applications that can use Setchain as the underlying decentralized infrastructure.

## References


[1] Ethereum roadmap - danksharding. https://ethereum.org/en/roadmap/danksharding/

[2] Matter labs - zksync era. https://github.com/matter-labs/zksync-era

[3] Ethereum development documentation - block size (2025), https://ethereum.org/en/developers/docs/blocks/#block-size

[4] Adams, H., Robinson, N., Zinsmeister, D., Keefer, M., Angeris, G.: Uniswap v3 core (2021), https://uniswap.org/whitepaper-v3.pdf

[5] Alakuijala, J., Szabadka, Z.: Brotli Compressed Data Format. RFC 7932 (Jul 2016). https://doi.org/10.17487/RFC7932, https://www.rfc-editor.org/info/rfc7932

[6] Barger, A., Manevich, Y., Meir, H., Tock, Y.: A byzantine fault-tolerant consensus library for hyperledger fabric (2021), https://arxiv.org/abs/2107.06922

[7] Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. Journal of Cryptographic Engineering **2**(2), 77–89 (Sep 2012). https://doi.org/10.1007/s13389-012-0027-1

[8] Blockcerts: Blockcerts: The open standard for blockchain credentials, https://github.com/blockchain-certificates

[9] Blocknative: Ethereum merge: The impact of proof-of-stake (2022), https://www.blocknative.com/blog/ethereum-merge-proof-of-stake

[10] Boitier, W., Del Pozzo, A., Garcia-Perez, A., Gazut, S., Jobic, P., Lemaire, A., Mahe, E., Mayoue, A., Perion, M., Rezende, T.F., Singh, D., Tucci-Piergiovanni, S.: Fantastyc: Blockchain-based federated learning made secure and practical. In: 2024 43rd International Symposium on Reliable Distributed Systems (SRDS). pp. 260–270 (2024). https://doi.org/10.1109/SRDS64841.2024.00033

[11] Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. CoRR **abs/1807.04938** (2018), http://arxiv.org/abs/1807.04938

[12] Buchnik, Y., Friedman, R.: Fireledger: a high throughput blockchain consensus protocol. Proc. VLDB Endow. **13**(9), 1525–1539 (May 2020). https://doi.org/10.14778/3397230.3397246

[13] Buterin, V.: Ethereum: A next-generation smart contract and decentralized application platform (2014), https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[14] Capretto, M., Ceresa, M., Fernández Anta, A., Russo, A., Sánchez, C.: Improving blockchain scalability with the setchain data-type. Distributed Ledger Technologies: Research and Practice **3**(2) (jun 2024). https://doi.org/10.1145/3626963

[15] Cason, D., Fynn, E., Milosevic, N., Milosevic, Z., Buchman, E., Pedone, F.: The design, architecture and performance of the tendermint blockchain network. In: 2021 40th International Sympo-

sium on Reliable Distributed Systems (SRDS). pp. 23–33 (2021). https://doi.org/10.1109/SRDS53918.2021.00012

[16] Castro, M., Liskov, B.: Practical byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. **20**(4), 398–461 (2002). https://doi.org/10.1145/571637.571640

[17] Chaum, D.L.: Computer systems established, maintained and trusted by mutually suspicious groups. Tech. Rep. UCB/ERL M79/10, Electronics Research Laboratory, University of California Riverside, CA, USA (22 February 1979)

[18] Crain, T., Natoli, C., Gramoli, V.: Red belly: A secure, fair and scalable open blockchain. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 466–483 (2021). https://doi.org/10.1109/SP40001.2021.00087

[19] Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient bft consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems. p. 34–50. EuroSys '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3492321.3519594

[20] Fabric, H.: Hyperledger fabric v3.0.0 release (2023), https://github.com/hyperledger/fabric/releases/tag/v3.0.0

[21] Fernández Anta, A., Georgiou, C., Herlihy, M., Potop-Butucaru, M.: Principles of Blockchain Systems. Morgan & Claypool Publishers (2021)

[22] Fernández Anta, A., Konwar, K., Georgiou, C., Nicolaou, N.: Formalizing and implementing distributed ledger objects. ACM Sigact News **49**(2), 58–76 (2018)

[23] Follow My Vote: Follow my vote. https://followmyvote.com/ (2026), accessed: 2026-02-25

[24] Fu, Y., Jing, M., Zhou, J., Wu, P., Wang, Y., Zhang, L., Hu, C.: Quantifying the blockchain trilemma: A comparative analysis of algorand, ethereum 2.0, and beyond (2024), https://ar5iv.labs.arxiv.org/html/2407.14335v1, accessed: 2024-10-01

[25] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: Proceedings of the 26th Symposium on Operating Systems Principles. p. 51–68. SOSP '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3132747.3132757

[26] Goderdzishvili, N., Gordadze, E., Gagnidze, N.: Georgia's blockchain-powered property registration: Never blocked, always secured: Ownership data kept best! In: Proceedings of the 11th International Conference on Theory and Practice of Electronic Governance. p. 673–675. ICEGOV '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3209415.3209437, https://doi.org/10.1145/3209415.3209437

[27] Gramoli, V., Guerraoui, R., Lebedev, A., Natoli, C., Voron, G.: Diablo: A benchmark suite for blockchains. In: Proceedings of the Eighteenth European Conference on Computer Systems. p. 540–556. EuroSys '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3552326.3567482, https://doi.org/10.1145/3552326.3567482

[28] Guo, B., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Dumbo: Faster asynchronous bft protocols. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 803–818. CCS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3417262

[29] Haber, S., Stornetta, W.S.: How to time-stamp a digital document. J. Cryptol. **3**(2), 99–111 (1991). https://doi.org/10.1007/BF00196791, https://doi.org/10.1007/BF00196791

[30] Josefsson, S., Liusvaara, I.: Rfc 8032: Edwards-curve digital signature algorithm (eddsa) (2017)

[31] Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 1353–1370. USENIX Association, Baltimore, MD (Aug 2018), https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

[32] Karmegam, A., Bianchi, G.L., Capretto, M., Ceresa, M., Fernández Anta, A., Sánchez, C.: Invited paper: Setchain algorithms for blockchain scalability. In: Bonomi, S., Mandal, P.S., Robinson, P., Sharma, G., Tixeuil, S. (eds.) Stabilization, Safety, and Security of Distributed Systems. pp. 287–302. Springer Nature Switzerland, Cham (2026)

[33] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: Omniledger: A secure, scale-out, decentralized ledger via sharding. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 583–598 (2018). https://doi.org/10.1109/SP.2018.000-5

[34] Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of bft protocols. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. p. 31–42. CCS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978399

[35] Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008), http://www.bitcoin.org/bitcoin.pdf, white paper

[36] National Institute of Standards and Technology (NIST): FIPS PUB 180-4: Secure Hash Standard (SHS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (Mar 2012), federal Information Processing Standards Publication

[37] News, M.: Digital diploma debuts at mit (2017), https://news.mit.edu/2017/mit-debuts-secure-digital-diploma-using-bitcoin-blockchain-technology-1017

[38] Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts (2017), https://plasma.io/plasma.pdf

[39] Poon, J., Dryja, T.: The bitcoin lightning network: Scalable off-chain instant payments (2016), https://lightning.network/lightning-network-paper.pdf

[40] Project, A.: Aragon: A platform for building decentralized organizations (2017), https://github.com/aragon/whitepaper

[41] Ranchal-Pedrosa, A., Gramoli, V.: Zlb: A blockchain to tolerate colluding majorities. In: 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 209–222 (2024). https://doi.org/10.1109/DSN58291.2024.00032

[42] Rao, I.S., Kiah, M.L.M., Hameed, M.M., Memon, Z.A.: Scalability of blockchain: a comprehensive review and future research direction. Cluster Computing **27**(5), 5547–5570 (Aug 2024). https://doi.org/10.1007/s10586-023-04257-7, https://doi.org/10.1007/s10586-023-04257-7

[43] Russo, A., Fernández Anta, A., González Vasco, M.I., Romano, S.P.: Chirotonia: A scalable and secure e-voting framework based on blockchains and linkable ring signatures. In: Xiang, Y., Wang, Z., Wang, H., Niemi, V. (eds.) 2021 IEEE International Conference on Blockchain, Blockchain 2021, Melbourne, Australia, December 6-8, 2021. pp. 417–424. IEEE (2021)

[44] Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: Dag bft protocols made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 2705–2718. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3559361

[45] Systems, I.: Malachite: Decentralize whatever, https://github.com/informalsystems/malachite

[46] Szabo, N.: Smart contracts: Building blocks for digital markets. Extropy **16** (1996)

[47] Wallet, A.: What is tps (transactions per second)? (2023), https://atomicwallet.io/academy/articles/what-is-tps

[48] Webisoft: Cosmos tps (transactions per second): Understanding the speed of the internet of blockchains (2023), https://webisoft.com/articles/cosmos-tps/

[49] Wiki, B.: Confirmation - bitcoin wiki (2023), https://en.bitcoin.it/wiki/Confirmation, accessed: 2024-10-01

[50] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. http://gavwood.com/paper.pdf

[51] Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. p. 347–356. PODC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293611.3331591, https://doi.org/10.1145/3293611.3331591

[52] Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: Scaling blockchain via full sharding. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. p. 931–948. CCS '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3243734.3243853, https://doi.org/10.1145/3243734.3243853

[53] Zhu, F., You, L., Wang, J., Li, L.: P-hotstuff: Parallel bft algorithm with throughput insensitive to propagation delay. Computer Networks **262**, 111183 (2025). https://doi.org/https://doi.org/10.1016/j.comnet.2025.111183, https://www.sciencedirect.com/science/article/pii/S1389128625001513