Invited Paper: Setchain Algorithms for Blockchain Scalability*

Arivarasan Karmegam[1,3], Gabina Luz Bianchi[6], Margarita Capretto[2,4], Martín Ceresa[5], Antonio Fernández Anta[2,1], and César Sánchez[2]

[1] IMDEA Networks Institute, Madrid, Spain
[2] IMDEA Software Institute, Madrid, Spain
[3] Universidad Carlos III de Madrid, Spain
[4] Universidad Politécnica de Madrid, Spain
[5] Input-Output, Madrid, Spain
[6] Universidad Nacional de Rosario, Argentina

Abstract. Setchain has been proposed to increase blockchain scalability by relaxing the strict total order requirement among transactions. Setchain organizes elements into a sequence of sets, referred to as *epochs,* so that elements within each epoch are unordered. In this paper, we propose and evaluate three distinct Setchain algorithms that leverage an underlying block-based ledger. *Vanilla* is a basic implementation that serves as a reference point. *Compresschain* aggregates elements into batches and compresses these batches before appending them as epochs in the ledger. *Hashchain* converts batches into fixed-length hashes, which are appended as epochs in the ledger. This requires Hashchain to use a distributed service to obtain the batch contents from its hash. To allow light clients to safely interact with only one server, the proposed algorithms maintain, as part of the Setchain, proofs for the epochs. 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 number of Byzantine servers assumed). All three Setchain algorithms are implemented on top of the CometBFT blockchain application platform. We conducted performance evaluations across various configurations, using clusters of four, seven, and ten servers. Our results show that the Setchain algorithms reach orders of magnitude higher throughput than the underlying blockchain, and achieve finality with latency below 4 seconds.

Keywords: Blockchain · Setchain · Scalability · CometBFT

1 Introduction

A blockchain is a *reliable distributed object* containing the list of transactions performed on behalf of users, totally ordered and packed into blocks [9,8]. Real-world

* 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), and grant DECO (PID2022-138072OB-I00).

blockchains are maintained by multiple servers (without a central authority) that leverage a *Byzantine-tolerant consensus algorithm* to order the transactions and that compute their effects [6,7].

Challenges. One of the main challenges for practical blockchains is to append transactions to the chain at a fast rate. This is called *throughput* and is measured in transactions per second (TPS). Throughput is constrained by multiple factors, including the speed at which the consensus is solved, network latency, and the computational cost of executing transactions to update or validate the blockchain state. To maintain efficiency and decentralization, blockchains often impose limits on block size or execution complexity, which further impact throughput. A second challenge is to reduce the *latency* of the blockchain, i.e., the time spent from when users send their transactions until those transactions are added to the blockchain. The latency in some blockchains can be of hours. Moreover, in some blockchains (e.g., Bitcoin), a block that has been reported to be added to the blockchain may end up being removed due to a fork in the chain. Hence, a third challenge is to achieve *finality* in the blockchain, i.e., reach a point in time when users know their transactions are in the chain, never to be removed. Finally, a fourth challenge is to provide users with *proofs* that their transactions are in the chain. This can be achieved by having each user contact several servers and ask them for individual proofs (e.g., a copy of the whole chain). This is rarely done, and users typically trust that the server with which they interact is honest. It would be desirable that solid proofs that the transactions are appended to the chain are provided to users who interact with one single server. We address these challenges with Setchain [4], which is a *reliable distributed object* that implements Byzantine-tolerant distributed grown-only sets with barriers and has been shown to have a large throughput, and employs *epoch-proofs,* which are cryptographic artifacts that allow users to interact reliably with just one server. Related Works section is deferred to the full version [13] due to space constraints.

Setchain. An approach to improve blockchain scalability is Setchain [4], a Byzantine-tolerant distributed object that implements a sequence of sets (called *epochs*). Setchain relaxes the total order requirement of blockchain and thus can achieve higher throughput and scalability by allowing transactions to be validated in parallel within an epoch. Setchain can be used for applications like digital registries (e.g., MIT Digital Diplomas [3,15], Georgia Government registry) or voting systems (e.g., Follow My Vote, Chirotonia [16]), where different elements in the blockchain need not be ordered except across infrequent epoch barriers. Byzantine-tolerant distributed algorithms that implement Setchain have been proposed, but no efficient real-world implementations exist.

Contributions. In this work, we propose a family of real-world Setchain algorithms built on top of a block-based ledger. We follow an incremental approach, providing several approximations to the final and most complex solution (Section 3). We first present **Algorithm Vanilla**, which is a naive implementation of Setchain with similar throughput as the underlying ledger. Then, we present **Algorithm Compresschain** that increases the throughput using compression of epochs. Finally, we present **Algorithm Hashchain**, a solution using

hash functions. From these algorithms, our primary contribution is Hashchain, which exploits the succinctness of hashes to reduce the communication necessary during broadcasts and consensus, communicating a fixed-size hash instead of the hundreds or thousands of elements of an epoch. The price to pay is an additional distributed algorithm to obtain the epoch contents of a hash from the corresponding server. Additionally, to allow users to interact with only one server, these algorithms maintain, as part of the Setchain, proofs for its epochs. An *epoch-proof* is the hash of the epoch, cryptographically signed by a server. A user can verify the correctness of an epoch with $f + 1$ epoch-proofs (where f is the maximum number of Byzantine servers). All three Setchain algorithms are implemented on top of the CometBFT blockchain application platform (Section 4). We deployed these implementations as Docker nodes in a cluster. We conducted performance evaluations across various configurations, using clusters of four, seven, and ten servers. Our results show that the Setchain algorithms reach orders of magnitude higher throughput than the underlying blockchain (tens of thousands TPS), and achieve finality with latency below 4 seconds.

2 Model and Definitions

System Model. We consider a distributed system consisting of n servers and an unbounded number of clients, which together are the system processes. The system is permissioned but public, also known as "open permissioned" [7], referring to an open model for clients but where servers are known upfront (permissioned). Nevertheless, this model can also be adapted to a permissionless setting with committee sortition [10] without significant modifications. Up to $f < n/2$ of the servers may exhibit Byzantine behavior, while the clients' behavior is not restricted (i.e., they can be Byzantine). We assume that the number f of Byzantine servers is known. We use $f+1$ as a lower bound on the number of consistent epoch-proofs required to ensure that an epoch is correct and as the number of signatures needed in order to consolidate hashes into epochs (see Section 3). We assume that there is a public key infrastructure (PKI) deployed so that each process (server or client) has a pair of private and public keys. Every process also knows the public key of all the other processes. The communication between processes is assumed to be reliable, meaning that messages sent between correct processes are eventually delivered only once, and no spurious messages are generated. Faulty processes can send any message, but thanks to the PKI, they cannot impersonate other processes. The PKI enables authenticated and secure communication between processes, since messages are signed by the sender using their private key, and the receiver verifies the signature using the sender's public key. If a signature is invalid, the message is discarded. Clients create elements and invoke a Setchain operation to add them. The elements created by the clients are signed and can hence be authenticated. They can also be validated by the servers for syntactic and semantic correctness. Only authenticated valid elements are processed by the correct servers. We assume that a server cannot create a valid element by itself, and that clients and servers do not collude.

Setchain. A Setchain [4] is a Byzantine-tolerant distributed object `S` that implements a sequence of sets called *epochs*. The Setchain defines an order between elements in different epochs, but not between elements in the same epoch. Thus, Setchain relaxes the total order requirement imposed by blockchains, potentially achieving higher throughput and scalability. Let U be the set of elements that client processes can inject into the Setchain. A Setchain `S` is a reliable distributed object where a collection of servers maintain: (i) a grow-only set $the_set \subseteq U$ of elements added; (ii) a natural number $epoch \in \mathbb{N}$; (iii) a map $history : \{1, \ldots, epoch\} \to 2^U$ with the set of elements that have been stamped with a given epoch number[7]; (iv) a set *proofs* of epoch-proofs (see Section 2)[8]. Servers support two operations: `add` and `get`. Operation $\texttt{S.add}_v(e)$ is used by a client to request a server v to add an element e to the Setchain `S`, while operation $\texttt{S.get}_v()$ returns the values $(the_set, history, epoch, proofs)$ that server v maintains. Servers can use a function $\texttt{valid_element}(e)$ to locally decide if an element e is valid. In a Setchain, when a new epoch is created, the servers collaboratively decide which of the added elements not yet assigned an epoch are included in the new epoch, and increase the epoch number. We call this an *epoch increment*. For the rest of the paper, we assume that at any given time, there will be a future epoch increment. This assumption is reasonable in real-world scenarios, and it can be reliably ensured by utilizing timeouts. A typical workflow from the point of view of a client is as follows: a client invokes $\texttt{S.add}_v(e)$ in a server v to insert a new valid element e in the Setchain. The element e will be propagated among the servers, and when an epoch increment occurs, the servers will attempt to include it in the new epoch. After waiting for some time, the client invokes $\texttt{S.get}_w()$ in a (possibly different) server w to check that the element has been effectively added to the Setchain and included in an epoch. Setchain implementations must satisfy certain properties that guarantee consistency between correct servers and that the valid elements added are eventually included in an epoch [4]. These properties are restricted to correct servers, since Byzantine servers do not provide any guarantee.

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 $i \in \{1, \ldots, h\}, H[i] \subseteq T$.
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 $e \in T$.
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 $e \in T'$.
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 $e \in H'$.

[7] 2^U denotes the power set of U.

[8] This set was not part of the API described in the original Setchain paper [4] and is one of the contributions of this work.

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, $H[i] \cap H[i'] = \emptyset$.
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 $H[i] = H'[i]$.
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.

Setchain Epoch-proofs. The original Setchain [4] proposal assumes that a client interacts with a correct server. However, in practice, clients do not know whether they are communicating with a correct or a Byzantine server. Hence, a client may need to interact with a sufficient number of servers (at least $f+1$) to guarantee that at least one is correct. In this work, we introduce *epoch-proofs* as a mechanism that allows clients to achieve the same guarantees without needing to contact multiple servers, thereby simplifying the process and improving efficiency. An epoch-proof is the cryptographic signature of an epoch i by a server v. As described above, in this paper, the Setchain maintains a set *proofs* of epoch-proofs. This set is returned by the Setchain when a `get` operation is invoked. Hence, when *proofs* contain at least $f + 1$ consistent epoch-proofs for a given epoch i, the client can be sure the epoch is correct. The epoch-proof is created by signing the hash of the epoch number and the elements of the epoch: $p_v(i) = \texttt{Sign}_v(\texttt{Hash}(i, \mathit{history}[i]))$. To add an element, a client only performs a single $\texttt{S.add}_v(e)$ request to one server v, hoping it is a correct server. After waiting for some time, the client can invoke a $\texttt{S.get}_w()$ from a single server w and check whether e is in some epoch, and the set *proofs* returned contains at least $f + 1$ valid epoch-proofs for that epoch to guarantee that at least one correct server has signed it. Clients can verify whether epoch-proofs are valid by generating the hash of a given epoch, and verifying if the signature in the epoch-proof is valid using the hash and the public key of the signing server. Recall that servers' public keys are known to the clients. Note that this process usually requires only one message per `add` and one message per `get`. Of course, it is possible that the client is unlucky and servers v or w are Byzantine, and after waiting a reasonable amount of time, has to restart the process. The following is the basic property that any Setchain algorithm must satisfy concerning epoch-proofs.

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 $H[i]$.

Block-based Ledger. In the proposed algorithms to implement a Setchain as presented in this paper, we assume the availability of a Byzantine-tolerant consensus service. For simplicity, we abstract that service as a block-based ledger. While our algorithms only require that the number of Byzantine processes satisfies $f < n/2$, the block-based ledger may have more stringent requirements. For instance, in Section 4 we use the ledger CometBFT that requires $f < n/3$. A

block-based ledger L is a Byzantine-tolerant distributed object that maintains a sequence of blocks. Each block contains a sequence of transactions that have been appended to the ledger by its clients. We prefer not to call this object a blockchain since its transactions have no semantics. To keep the nomenclature consistent, the term *transaction* is always used to refer to *block-based ledger transactions*, while *element* refers to the elements of the Setchain. Depending on the implementation, a single ledger transaction may contain one or many elements. The block-based ledger provides two endpoints. First, $\texttt{append}(tx)$ is used to submit a transaction tx to the ledger, which eventually gets included in a block. Then, $\texttt{new_block}(B)$ notifies the servers whenever a new block B has been appended. The number of transactions in B is $|B|$, and the ith transaction in B is $B[i]$. Since Setchain algorithms use a block-based ledger, to prove correctness, it is necessary to define properties that the block-based ledger guarantees. In particular, the ledger property that will be used is that any valid transaction appended by a correct server will eventually be included in a block, which is then notified to all the servers. Moreover, we assume that all blocks notified are final.

9. *Ledger-Add-Eventual-Notify:* Let tx be a valid transaction, and let v be a correct server. Then, if v invokes $\texttt{L.append}_v(tx)$, 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 $\texttt{L.new_block}_w(B)$.
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.
11. *Notification-Implies-Append:* If a correct server w is notified with $\texttt{L}.new_block_w(B)$ containing a valid element e, then some server v had invoked $\texttt{L.append}_v(e)$.

3 Setchain Algorithms

In this section, we propose practical Setchain algorithms built on top of a block-based ledger. In particular, we present three different implementations, beginning with a naive but trivially correct implementation and ending with a more complex algorithm implementing Setchain using hashing. As described in Section 2, a Setchain S provides two methods, `add` and `get`, defined in each algorithm.

Vanilla. In Vanilla, we propose a basic implementation of Setchain S using a block-based ledger L. The algorithm for Vanilla is deferred to the full version [13] due to space constraints. In this algorithm, each server v maintains three local sets: *the_set*, *history*, and *proofs*, and the counter *epoch*, as defined in Section 2. Whenever a client queries Setchain S using $\texttt{S.get}_v()$, the server returns the current value of *the_set*, *history*, *epoch*, and *proofs* (Line 9). On the other hand, clients add 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 so, the element is added to *the_set*, and the server invokes a $\texttt{L.append}_v(e)$ operation, which eventually adds the element e to ledger L. Whenever a new block is appended to the ledger

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
```

`L`, server v is notified through `L.new_block`(B) (see Line 10). Ledger `L` is used to disseminate the epoch-proofs in the three algorithms proposed. Hence, the server starts by extracting the valid epoch-proofs from B and adding them to the set *proofs* (Line 12). Then, the server extracts the elements from B that are valid and not yet in an epoch, and places them into batch G (Observe that checking whether an element is valid cannot be avoided because a Byzantine server may have added invalid elements to the ledger). The set G of valid elements is added to *the_set*, and is used to form a new epoch. This new epoch is added to the *history* map. Then, the epoch-proof containing the epoch number, the epoch signature, and the server's identity is added to the ledger by invoking an `L.append` operation (Line 18). Vanilla solves the proof challenge via epoch-proofs, allowing a client to interact with only one correct server. However, the throughput and latency of the Setchain implemented are those of the block-based ledger `L`.

Compresschain. The second algorithm we propose, Compresschain, increases the throughput compared to Vanilla. In Compresschain, each server has a *collector* in which the elements added by clients and epoch-proofs added by servers are held until the collector size is reached (or possibly a timeout expires). Then, the collected batch, which will become a new epoch, is compressed and appended to the ledger as a single ledger transaction. Compresschain adapts Vanilla to handle the compressed batches by adding an additional set *batch* to collect client elements and epoch-proof. When a client adds an element e to the Setchain using `S.add`$_v(e)$, in addition to adding the element to *the_set*, it is also added to *batch* (the collector; Line 5). When *batch* reaches the collector size (or a timeout is triggered, and the *batch* is not empty), a notification `isReady`$(batch)$ is signaled. Then, the batch is compressed and appended to the ledger, and *batch* is reset to $\emptyset$. When the creation of a new block B is notified,

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) 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 = ∅) ∨   ▷ Not found
        (Hash(batch_original) ≠ h) then
29     continue
30    hash_to_batch[h] ← batch_original
31    Register_batch(h, batch_original)
32    s_v ← Sign_v(h)
33    hb ← ⟨h, s_v, v⟩
34    L.append(hb)
35    np ← {ep ∈ batch_original :
        ep = ⟨j, p, w⟩ is epoch-proof ∧
        valid_proof(j, p, w, history[j])}
36    proofs ← proofs ∪ np
37    G ← {e ∈ batch_original :
        e is an element ∧ valid_element(e)
        ∧(e ∉ history)}
38    the_set ← the_set ∪ G
39    hash_to_signers[h] ← hash_to_signers[h]
        ∪{w}
40    if |hash_to_signers[h]| = f + 1 then
41     epoch ← epoch + 1
42     G ← {e ∈ batch_original :
        e is an element ∧ valid_element(e)
        ∧(e ∉ history)}
43     history[epoch] ← G
44     p ← Sign_v(Hash(epoch, G))
45     add_to_batch(⟨epoch, p, v⟩)
46 end upon
```

each compressed batch in B is processed in order. Let $B[i]$ be the ith transaction in block B. $B[i]$ is decompressed, and the valid epoch-proofs in $B[i]$ are added to the set *proofs*. Then, the valid elements in $B[i]$ are extracted to a set G to form a new epoch. The elements in G are first added to *the_set*. Then, they are assigned an epoch number, and the epoch information is added to the *history* map. An epoch-proof for the epoch is also created and sent to the collector using $\texttt{S.add_to_batch}_v(\cdot)$, which in turn adds it to the *batch*. The most important difference between Compresschain and Vanilla is that in Compresschain each transaction in a block B is a compressed batch (which potentially has many Setchain elements) that becomes an epoch, whereas in Vanilla, each transaction in block B is an element, and an epoch is the set of valid elements in the block B. This potentially increases the throughput significantly.

Hashchain. Hashchain increases the throughput by using hashing of batches instead of compressing them. While the space reduction of hashing may be enormous, hashes are irreversible. Hence, a non-trivial method has to be provided to recover the contents of the original batch of elements from a hash. In Hashchain, when a batch is ready (Line 12), server v generates the hash h of the batch, signs the hash h, and creates the hash-batch hb as the combination of the hash of the batch, the signature, and its identity. Then, the hash-batch hb is appended as a

transaction to the ledger L. As the original batch cannot be recovered from the hash h, the server saves the batch associated with h in a map *hash_to_batch*. Also, the *batch* with its hash h is registered using `Register_batch`. This will allow serving the contents of the original batch to other servers. The *batch* is then reset to $\emptyset$. Upon a `L.new_block`(B) notification, each valid hash-batch $hb_w = \langle h, s_w, w\rangle \in B$ is processed as follows. If server v does not have the batch associated with the hash h in the map *hash_to_batch*, v requests the batch from server w using `Request_batch` (which must have it since w signed that hash-batch). Since w may be Byzantine, server v waits for a limited amount of time for the answer to its request. If the batch is received correctly, v creates its hash-batch $hb_v = \langle h, s_v, v\rangle$ and appends it to ledger L. Then, the batch is saved in map *hash_to_batch* and *the_set* is updated. Observe that finding a valid hash-batch in ledger L is not enough to assign it an epoch number, because it could be a hash-batch appended by a Byzantine server that refuses to provide the batch that corresponds to the hash. Hence, we decided that a hash has to be signed by at least $f+1$ individual servers to be consolidated into an epoch. The reason is that a hash with hash-batches from $f+1$ different servers is guaranteed to be signed by at least one correct server, which will have a copy of the batch and serve it upon request. Then, the processing of hash-batch hb_w is continued by adding w to the set of signers in the map *hash_to_signers*, used to track the set of servers that appended hash-batches with hash h (Line 39). When this set of signers reaches $f+1$, the server extracts the valid elements from the batch to a set G which is assigned an epoch number. We call this process *epoch consolidation*. Also, the valid epoch-proofs in *batch_original* are added to the set *proofs*. As usual, the epoch is added to the *history*, and an epoch-proof for this epoch is generated and sent to the collector using $\texttt{S.add_to_batch}_v(ep)$.

Correctness Proof Sketch. We provide here an intuitive explanation for why the Setchain algorithms presented in Section 3 behave correctly, even in the presence of up to f Byzantine faults[9]. The Setchain algorithms ensure that all valid elements are eventually included in an epoch, and that all correct servers maintain a consistent view of this sequence. When a client submits an element to a Setchain server v, it is added to that server's local *the_set* only if the element is valid. Once stored locally, the element is returned as a part of that Setchain server's *the_set*, whenever $\texttt{S.get}_v()$ is invoked by the client (Property 2). When it consolidates an epoch, a server adds all valid elements from the epoch to *the_set*, if not already present, before recording the epoch in the *history*. Therefore, *history*, which is the collection of all the epochs, is a subset of *the_set* (Property 1). Each server periodically creates a batch either after collecting enough elements or after a timeout. Since all valid elements in the batch are added to *the_set* and eventually put into an epoch, these elements eventually end up in *history*, again by Properties 9 and 10 (Property 4). In Hashchain, a hash-batch consolidates into an epoch (i.e., is assigned an epoch number) only after the server receives $f+1$ signatures on the hash of the batch. When a correct server v either generates a hash-batch or receives a hash-batch from another server, it

[9] We defer the full proof of correctness to the full version [13] due to space restrictions.

first verifies the batch elements (requesting them from the signer of that batch, if the original batch is not found locally), then signs the batch hash and appends it to the ledger. Eventually, a block containing this hash-batch signed by v is notified to all the servers, and the other correct servers request the original batch from v (if the original batch is not found locally), verify the batch elements, sign the batch hash and append it to the ledger. So, given that there are at least $f + 1$ correct servers, eventually this hash-batch receives $f + 1$ signatures and the batch consolidates into an epoch. All correct Setchain servers use the same logic to construct epochs from observed blocks, all Setchain servers observe the same blocks in the same order, and all the transactions inside the block are totally ordered (Properties 9 and 10). Hence, correct Setchain servers create identical epochs and agree on the epoch content (Property 6). Also, each correct server ensures that new epochs exclude elements already included in *history*, which leads directly to Property 5. Again, by Property 9, every element added to a server's local *the_set* eventually appears in a block that is delivered to all Setchain servers. After that, every `S.get()` invocation to a correct Setchain server will include this element as part of their *the_set* (Property 3). Since we assume that a server can neither create a valid element by itself nor collude with clients, a server cannot append a valid element with `L.append`(e) without a client invocation of `S.add`(e) (Property 7). To ensure that clients can validate the correctness of an epoch without needing to trust a single server, Setchain maintains epoch-proofs, which are Setchain signatures of the hash of the epoch. These proofs are included in the ledger as transactions, enabling any client to retrieve them through a `S.get()` operation. When a correct Setchain server encounters an epoch, after validation, it signs the hash of the epoch and appends it to the block-based ledger as its epoch-proof. Due to Property 9, this epoch would have been notified to all Setchain servers and eventually the ledger contains $f + 1$ epoch-proofs for the epoch, as the number of correct servers is more than $f + 1$. Therefore, eventually, when a client invokes `S.get()`, at least $f + 1$ epoch-proofs are returned for an epoch (Property 8) and the client can trust the epoch.

4 Implementation and Performance Evaluation

The three algorithms presented have been implemented in Golang using CometBFT as the block-based ledger. We describe some implementation aspects.

CometBFT. CometBFT [5] (previously known as Tendermint) is a Byzantine-tolerant state machine replication engine. CometBFT is a blockchain middleware that supports replicating arbitrary applications, written in any programming language. More details about CometBFT are given in the full version [13].

Performance Evaluation Platform. We carried out the 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 used Docker Engine version 20.10.5. The Setchain algorithms are implemented on CometBFT v0.38, with each ledger server running in a Docker container and each container running on a separate machine in

the cluster. The containers have no limit on CPU or RAM usage. Each Docker container contains one client, one collector module, and one CometBFT server.

Experiment Scenarios. The parameters considered for the experiments in this work are listed in the full version [13]. Each client adds at a rate *sending_rate per server_count* by sending the elements to their local server (the server running in the same Docker). The *network_delay* parameter is an artificial latency that is added to all communications between servers to simulate the impact of moving from a cluster to a wide area deployment. The mempool is an important element of CometBFT, where the unconfirmed ledger transactions are held after validation and before being included in the blockchain by CometBFT. The default mempool setting of CometBFT allows only a maximum of $5,000$ transactions to be present in the mempool. For the Setchain evaluation, we did not want this to be a bottleneck. Hence, after a few trial and error experiments, the mempool size has been set to $10,000,000$ transactions or 2 GB, whichever is reached first. By default, each experiment has been designed so that clients add elements in the Setchain for 50 seconds. When all the elements have been included in epochs and all the epoch-proofs have been inserted in ledger blocks, the experiment ends. The logs are then collected and analyzed. The injection and throughput metrics are based on the Setchain elements sent by the clients, measured in *elements per second (el/s)*. To keep the experiments realistic, we use transactions downloaded from Arbitrum [12] as Setchain elements. For hashing, we use SHA512 [14]; signing is done with ed25519, which is part of the EdDSA (Edwards-Curve Digital Signature Algorithm) family [11,2], and we use Brotli for compression [1]. The average size of an Arbitrum transaction is approximately 438 bytes with a standard deviation of 753.5. The length of an epoch-proof is 139 bytes. In Compresschain, the average length of a compressed batch is approximately $16,000$ bytes with a standard deviation of $2,100$ for a collector limit of 100, and an average of $66,000$ bytes with a standard deviation of $15,000$ for a collector limit of 500. So, the compression ratio varies approximately from 2.5 to 3.5. In Hashchain, the length of a hash-batch is 139 bytes. In CometBFT, one ledger block is generated roughly every 1.25 seconds (i.e., the block rate is approximately 0.8 blocks/s). Unless otherwise stated, the ledger block size used in CometBFT is 0.5 MB.

Analysis. We derived an analytical formula to estimate the theoretical highest throughput of each algorithm as a function of system parameters. These formulas are used in this section to compute the theoretical throughput values, which are then compared with our experimental results. Due to space constraints, the full derivation is deferred to the full version [13].

Throughput Comparison. Fig. 1 shows the throughput (in elements *committed* per second, el/s) over time of the three Setchain algorithms for different sending rates with 10 servers and with no increase in the network delay. An element added becomes *committed* when the epoch in which the element has been included gets at least $f + 1$ epoch-proofs in the ledger. The throughput values obtained in the analysis are also shown for reference. In Fig. 1 (left), for a sending rate of $5,000$ el/s and collector size 100, it can be observed that both Vanilla

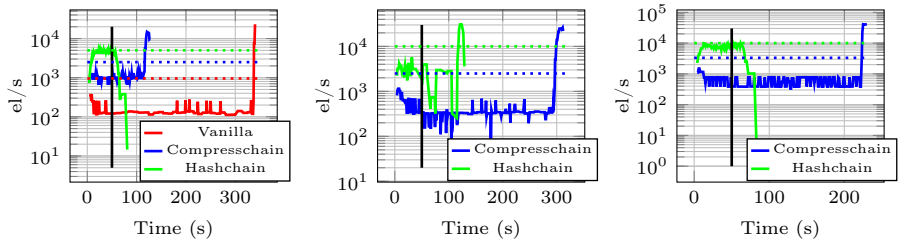


Fig. 1: Throughput over time of the Setchain algorithms for (left) Sending rate = 5,000 el/s and Collector size = 100, (center) Sending rate = 10,000 and Collector size = 100, and (right) Sending rate = 10,000 el/s and Collector size = 500. Solid lines plot the rolling average number of elements committed 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 the analysis section available in the full version [13].

and Compresschain take a long time to commit all the elements and show the peak at the end, which is a symptom of stress. Moreover, both algorithms show a smaller throughput than the analytical bound. Hashchain, on the other hand, is able to cope and finishes shortly after all elements are added. In Fig. 1 (center), for a sending rate of 10,000 el/s and collector size 100, we have excluded Vanilla to compare Compresschain and Hashchain more efficiently. In this plot, it can be seen that both Compresschain and Hashchain are stressed, although the former is stressed much more than the latter. As shown in Fig. 1 (right), increasing the collector size to 500 helps to relieve the stress from Hashchain, while it does not help Compresschain much. We present the average throughput achieved up to 50 s for all three algorithms in every experiment in Fig. 1, in a table in the full version [13].

Pushing the Hashchain Limits. As seen in Fig. 1 (right), there is a noticeable gap between the analytical achievable throughput (147,857 el/s, see full version [13]) and the actual throughput observed with the implementation of Hashchain, because the experiments did not try large sending rates. To determine the highest achievable throughput, we increased the sending rate. However, we identified a bottleneck around 20,000 el/s, regardless of further increase in the sending rate beyond that threshold and of the collector size (as seen in Fig. 2 (left)). The most likely cause of this limitation is the hash-reversal process, where batches are exchanged between servers across the network for each batch-hash sent by the collector. In our implementation, Setchain servers handle the distribution of transaction batches. More efficient methods could be employed, such as having only a set of $2f + 1$ servers sign each batch-hash and epoch, utilizing optimistic validation of hash-batches (share a batch only on request), or implementing alternative distributed batch-sharing mechanisms. To assess the impact of hash-reversal, we conducted experiments removing this service and the validation of hash-batches, while assuming that all servers are correct—meaning all hash-batches are inherently valid. Fig. 2 (left) illustrates the highest achieved throughput by Hashchain with and without hash-reversal. From this,

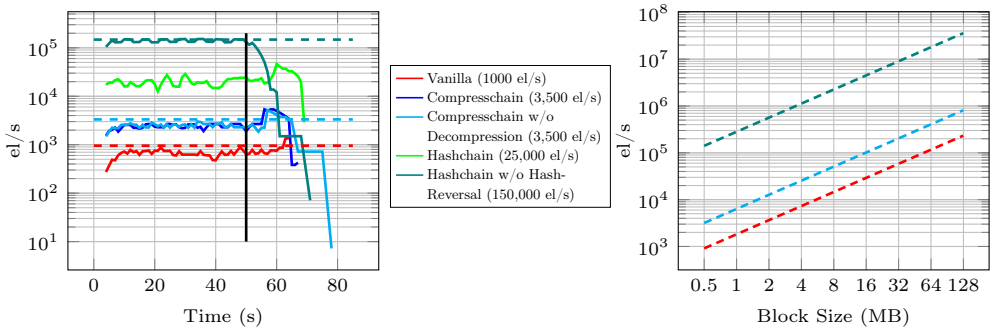
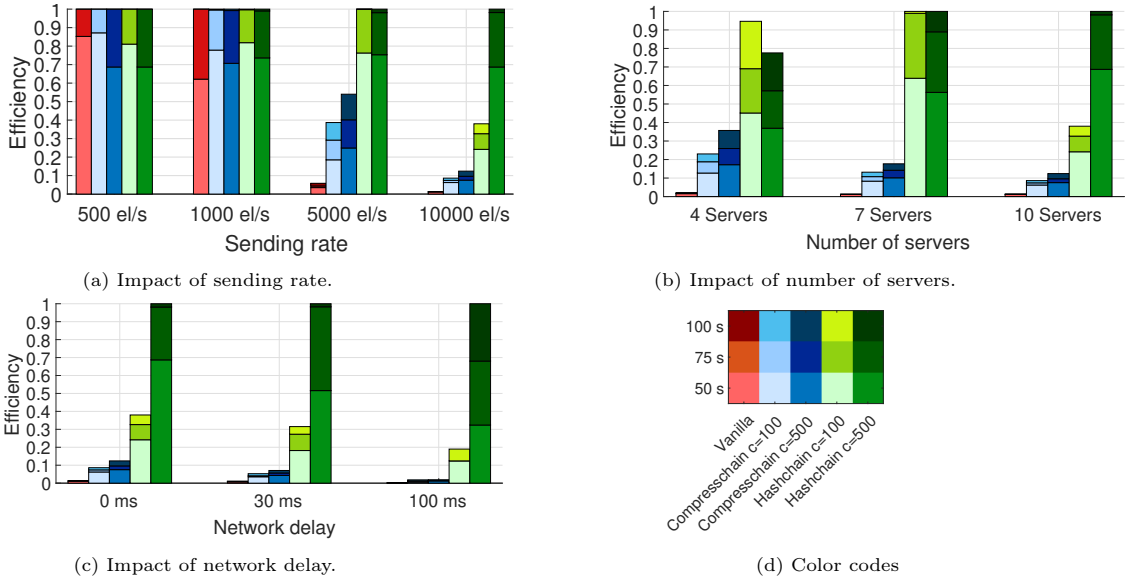


Fig. 2: (left) Highest throughput measured in the evaluation platform with collector size 500. The vertical bar marks the time clients add the last element (roughly after 50 s). Solid lines plot the rolling average number of elements committed in 9 s. The dashed lines show the analytical throughput. (right) Analytical throughput of the Setchain algorithms for various block sizes with collector size 500.

it is clear that hash-reversal is the bottleneck, and with a more efficient implementation of hash-reversal, Hashchain would scale much better in throughput. The results confirm that hash-reversal significantly limits performance. Without it, Hashchain reaches an average throughput of 133, 882 el/s for the first 50 seconds with a sending rate of 150, 000 el/s, compared to an average throughput of 20, 061 el/s for the first 50 seconds with a sending rate of 25, 000 el/s with hash-reversal enabled. It is also important to note that these results were obtained with a collector size of 500, chosen to maintain comparability with other experiments in this work. A larger collector size would likely yield even higher throughput without hash-reversal. For comparison, Fig. 2 (left) also shows the highest achieved throughput by Compresschain and Vanilla. For Compresschain, we ran it with and without decompression and validation to measure the impact of these processes. Observed throughputs are well below Hashchain's throughput, even with hash-reversal. The highest throughputs observed with Vanilla, Compresschain Light, and Hashchain Light are very close to the analytical values. Fig. 2 (right) shows the analytical throughput for all three Setchain algorithms for larger block sizes, while keeping the other parameters constant (For Compresschain and Hashchain, Collector size = 500). As can be seen, with the usual 4MB block size of CometBFT, Hashchain reaches a throughput of 10^6 el/s, and with blocks of 128 MB reaches more than 30 million el/s.

Efficiency. To easily quantify the level of stress of an algorithm, we define and use a metric that we call *efficiency*. The efficiency is obtained by dividing the number of elements committed by the total number of elements added. We calculate the efficiency factor after 50, 75, and 100 seconds. Remember that clients add elements for 50 seconds in every experiment. If the algorithm is not stressed, we expect to observe efficiency close to 1 after 50 seconds, and exactly 1 after 75 seconds. In this section, we use a base scenario with 10 servers, a sending rate of 10, 000 el/s, and no (0) network delay, and vary one of these parameters at a time. Fig. 3a shows the efficiency for four different sending rates (500, 1000,



(a) Impact of sending rate.

(b) Impact of number of servers.

(c) Impact of network delay.

(d) Color codes

Fig. 3: 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.

5,000, and 10,000), and for collector sizes 100 and 500, when applicable. This figure shows that all algorithms reach full efficiency in 70 seconds for the lower rates 500 and 1,000. Then, for rates 5,000 and 10,000, Vanilla has very low efficiency. Compresschain, on the other hand, also reduces its efficiency significantly, and increasing the collector size from 100 to 500 does not help much. Finally, Hashchain only shows a decrease in efficiency with a rate of 10,000, which is alleviated by increasing the collector size. In Fig. 3b, we compare the change in efficiency observed when the number of servers is varied, maintaining a sending rate of 10,000 el/s. Observe that Vanilla has the lowest efficiency, even for 4 servers. The efficiency of Compresschain is also low, and increasing the collector size does not improve it much. Moreover, it decreases as the number of servers increases. Finally, Hashchain only shows low efficiency with 10 servers and a collector size of 100. Interestingly, it shows a less-than-perfect efficiency for 4 servers, which may be due to having fewer servers available for the reverse hashing process. Fig. 3c shows how adding an artificial delay to all communication messages affects efficiency. This models the impact of moving from a local-area network to a wide-area network. As can be observed, the increase in network delay reduces the efficiency. However, even with the largest delay of 100 ms, Hashchain with a collector size of 500 achieves full efficiency in 100 seconds.

Latency. When a valid element is sent by a client to its Setchain server, it is eventually sent in a ledger transaction to a CometBFT server. The transaction, once validated, enters the mempool of the CometBFT server. Then, it is shared with other CometBFT servers via a gossip protocol. These servers validate the transaction and replicate it in their local mempool. Eventually, the transaction will be included in a ledger block, and the Setchain servers will send epoch-

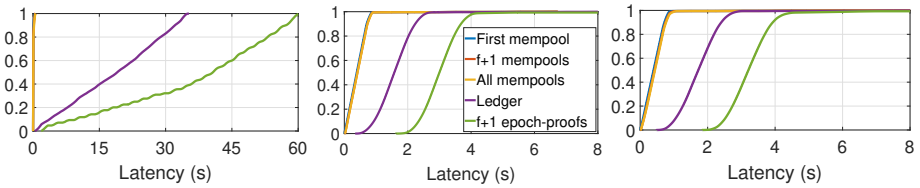


Fig. 4: Cumulative distribution function $F(x)$ of the latency experienced by the elements added to the Setchain to reach several stages in their process for (left) Vanilla, (center) Compresschain and (right) Hashchain. The collector size is 100. The scenario is with 10 servers, a sending rate of $1,250$ el/s, and no network delay.

proofs to the ledger. When $f+1$ such proofs have been included in ledger blocks, the element has been committed. Fig. 4 shows the distribution of the latencies experienced by the elements added to the Setchain to reach five different stages until commit, for each algorithm. It shows the latency until an element reaches (1) a CometBFT server mempool, (2) $f+1$ CometBFT server mempools, (3) all the CometBFT server mempools, and (4) the ledger. Finally, it shows the (5) commit latency (when $f+1$ epoch-proofs of the epoch in which the element was included are in the ledger). In Fig. 4 (left), the mempools are reached almost immediately because in Vanilla, the elements are directly sent to the CometBFT server. Compresschain and Hashchain, on the other hand, wait for the collector to be full or a timeout before sending the batch of elements to the CometBFT server as a transaction. This explains the delay in mempool latency in Fig. 4 (center) and (right). Consider now the latency an element takes to reach the ledger and to be committed (with $f+1$ epoch-proofs). With Vanilla, these two time gaps are tens of seconds in most cases, while for Compresschain and Hashchain, they are usually between one and two seconds. These two algorithms show in this scenario commit latency and finality below 4 seconds with probability almost 1.

5 Conclusion and Discussions

In this work, we presented three real-world Setchain implementation algorithms built on top of a block-based ledger. We have formally verified that the properties of Setchain hold for these algorithms (proofs are deferred to the full version due to space limitations [13]). We have implemented the three algorithms and performed an empirical evaluation to evaluate the effectiveness of each approach in various scenarios. Among the three, the Hashchain algorithm consistently performed better across all evaluated metrics. Its efficient use of hashing techniques allowed for improved scalability, making it particularly suitable for large-scale applications. Compresschain, while offering some improvements over Vanilla, did not match the performance gains observed with Hashchain.

References

1. 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

2. 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
3. Blockcerts: Blockcerts: The open standard for blockchain credentials, https://github.com/blockchain-certificates
4. 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
5. 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 Symposium on Reliable Distributed Systems (SRDS). pp. 23–33 (2021). https://doi.org/10.1109/SRDS53918.2021.00012
6. 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
7. 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
8. Fernández Anta, A., Georgiou, C., Herlihy, M., Potop-Butucaru, M.: Principles of Blockchain Systems. Morgan & Claypool Publishers (2021)
9. Fernández Anta, A., Konwar, K., Georgiou, C., Nicolaou, N.: Formalizing and implementing distributed ledger objects. ACM Sigact News **49**(2), 58–76 (2018)
10. 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
11. Josefsson, S., Liusvaara, I.: Rfc 8032: Edwards-curve digital signature algorithm (eddsa) (2017)
12. 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
13. Karmegam, A., Bianchi, G.L., Capretto, M., Ceresa, M., Fernández Anta, A., Sánchez, C.: Setchain algorithms for blockchain scalability (2025), https://arxiv.org/abs/2509.09795
14. 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
15. News, M.: Digital diploma debuts at mit (2017), https://news.mit.edu/2017/mit-debuts-secure-digital-diploma-using-bitcoin-blockchain-technology-1017
16. 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)