CAT: Concurrent Attestation of Transaction Execution in the Stateless Blockchain (Technical Report)

Anonymous Authors
placeholder
placeholder, placeholder
placeholder

ABSTRACT

Blockchain as an emerging technology has attracted much attention from both industry and academia. To overcome the intrinsic heavy computation and storage burden issues, the concept of stateless blockchain is proposed where only a cryptography commitment to system states is maintained on-chain and validators update the commitment based on transactions and corresponding correctness attestation (*i.e.*, witnesses/proofs). However, such a design still requires sequential transaction processing which restricts the system throughput and scalability. Meanwhile, due to the latency of witnesses/proofs generation, the commitment that the proofs are generated on may already have been updated which invalidates the proofs of under-processing transactions causing a high abort rate.

To tackle these issues and support general-purpose transactions, in this paper, we propose CAT which is a novel stateless paradigm to support concurrent transaction processing. In particular, we explore the execute-order-validate (EOV) transactions processing model and the idea of optimistic concurrency control (OCC) to deal with the concurrency. Besides, we perform sharding on the state commitment and design a novel data structure named commitment forest to maintain the commitment. In addition, since an optimal transaction packing strategy can further improve the parallelism of CAT, we propose an optimization problem named MinSC and prove its NP-hardness. A dynamic programming-based exact solution and a constant-factor approximation algorithm are proposed to implement parallel on both small and large scales. We apply CAT on a well-known stateless blockchain named zkSync. Experiments show that CAT can improve 61.2% effective throughput and 17.3% latency on average compared to the original design of zkSync.

KEYWORDS

Stateless; Concurrency Control; Optimization;

1 INTRODUCTION

Blockchain is an immutable ever-growing hash chain of transaction blocks providing a decentralized collaboration environment for users without a certain degree of trust. Specifically, users send their transactions to the network, which are verified and recorded on-chain through consensus protocols (*e.g.*, PoW [34] in permissionless chains where any node can join or leave and PBFT [11] in permissioned chains where node identities are known by each other) by *validators* (*e.g.*, miners in Bitcoin [34] and validation nodes in Fabric [5]). Following the original UTXO-based transaction model of Bitcoin, Ethereum [42] introduces the account-based model to extend transactions to support more generalized functionalities, including realizing the concept of *smart contract*, a program describing the business logic. However, well-known problems such as the limited throughput caused by the consensus bottleneck, ever-growing storage burden, and high computational cost for validators bring severe limitations to blockchain applications [17, 18, 43].

Recently, the concept of *stateless blockchain* is proposed to address the aforementioned issues [9, 40]. In particular, in the account-based model, system states are in the key-value fashion and updated by transactions. To be stateless, all system states are organized in an accumulator [19] (*e.g.*, Merkle tree [32], RSA accumulator [10], vector commitment [28], etc.). Meanwhile, only a *cryptography commitment* (*e.g.*, Merkle hash root) representing the digest of the accumulator with current system states is maintained on-chain.

In this case, transaction can be executed off-chain. Meanwhile, validators do not need to store heavy ledger data except the latest state commitment, preventing them from becoming the system bottleneck. Specifically, some nodes (namely *proposers*) execute transactions to obtain an updated state commitment reflecting the state transition result. Then, with techniques such as *zero-knowledge proof (ZKP)* [7] and *trusted execution environment (TEE)* [37], proposers can generate a proof of the transactions execution correctness, leading to the commitment update. Finally, validators only need to verify the proof and then record the updated state commitment on-chain by running a consensus protocol.

Although existing stateless designs reduce the burden of validators and apportion part of transaction processing workloads off-chain, they still suffer from several practical issues:

Firstly, it is time costly to generate the execution correctness proofs. As studied in [27], it takes hundreds of seconds to create proofs for 10k operations. Things are even worse when zero-knowledge proof and verifiable computing are applied. Zcash [38] as a stateless version of Bitcoin, takes few minutes to prove correctness for one transaction and zk-Rollup [20], a stateless scaling solution for Ethereum, takes tens of minutes to create proofs for one batch with hundreds of transactions [45]. Therefore, the transaction processing latency becomes a bottleneck. Additionally, a block with high commit latency hinders the processing of subsequent blocks.

Secondly, the transaction processing is forced to be serialized. Because the proof is tied to the commitment value that the transactions are executed on, and committed transactions will update both system states and the commitment value. In particular, existing solutions deal with the concurrency issue that different transactions are processed based on the same state commitment value in two ways: 1) let validators commit one of them and discard others even if there is no conflict on their accessed states, and let proposers regenerate proofs (example is EDRAX [13]). This solution limits the effective system throughput and wastes proposers' computation

resources. 2) rely on a centralized server to coordinate the transaction execution and proof generation (example is zkSync [30]). This mechanism breaks the decentralization property of blockchain.

To ease these issues, we need a novel stateless design, achieving proposing transaction execution results concurrently and correctness attesting without a centralized coordinator. **To achieve this goal, we need to address the following three challenges:**

(1) Decomposing the high dependency of transaction execution attestation and commitment value update addresses the first issue. Because the transaction processing can work in the pipeline, reducing the average processing latency.
(2) Enabling parallel transaction execution results proposing and attestation to improve throughput addresses the second issue. Specifically, with asynchronously proposed execution results, a concurrency control mechanism is required for validators to handle potential conflicts. In addition, for more comprehensive applications, the mechanism needs to support general-purpose transactions[1] which may read/write arbitrary states.
(3) Optimizing the design maximizes system parallelism and reduces the transaction abort rate for higher effective throughput with the concurrency correctness guarantee.

In this paper, we propose a novel paradigm named CAT which can be applied to both permissioned and permissionless stateless chains to support concurrent transaction process. **To solve the first challenge**, we leverage the sharding technique on the state commitment and borrow the idea of execute-order-validate (EOV) transaction processing model. Specifically, the framework works in a pipeline that multiple proposers asynchronously execute transactions off-chain and prewrite the execution results on-chain without an execution correctness proof. Then, corresponding proofs are generated by proposers and validated by validators to commit the transactions and finalize updated commitments. **To address the second challenge**, we propose a novel data structure named *commitment forest* to maintain the commitments accessed and updated by general-purpose transactions with concurrency control support. Besides, our method does not tamper with the consensus layer, providing usability for both permissioned and permissionless chains. **To capture the third challenge**, we model an optimization problem named *MinSC*, aiming to provide a strategy for proposers to process transactions in their local pools to maximize the system parallelism. We prove its NP-hardness and propose both a dynamic programming-based exact solution and an approximation algorithm with the approximation ratio of $1 + \frac{1}{e}$.

We summarize our contributions as follows:

(1) We propose CAT, a novel transaction processing framework for the account-based stateless blockchain which supports concurrent state commitment update.
(2) We design a data structure named commitment forest to maintain the state commitment in the account-based model to support general-purpose transactions, which also helps to coordinate the concurrent transaction execution result proposing for proposers and correctness attestation for validators.
(3) We model an optimization problem named MinSC for proposers to process transactions in their local pools to maximize the system parallelism in CAT. Besides, we prove the NP-hardness of MinSC and propose an exact solution and a constant-factor approximation algorithm to address it.
(4) We apply CAT on a real stateless system zkSync [30] and conduct extensive experiments to evaluate the actual performance of the CAT framework and our optimization algorithms. Results show that CAT, with our optimization, can improve 61.2% effective throughput and 17.3% latency on average compared to the state of art work, the vanilla zkSync.

The rest of paper is organized as follows: in Sec. 2 we introduce the background and preliminary concepts. We introduce the CAT framework in Sec. 3 and propose the commitment forest and its usage in Sec. 4. We propose batch selection optimization with solutions in Sec. 5. We analyse the correctness, attack model, liveness and storage cost of CAT in Sec. 6. We discuss the experimental results in Sec. 7 and related works in Sec. 8. We conclude in Sec. 9.

2 BACKGROUND AND PRELIMINARY

In this section, we introduce the background and some preliminary concepts that are needed in this work as well as related works.

2.1 Account-based Stateless Blockchain

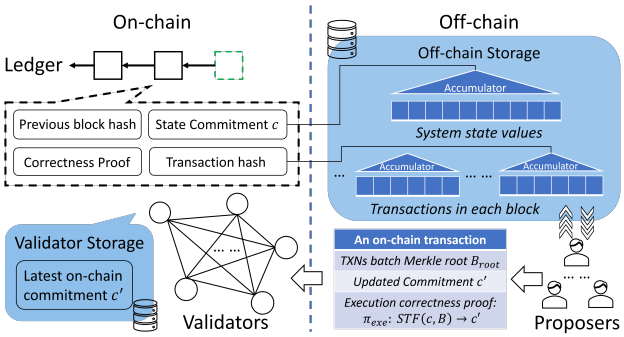


Figure 1: Basic architecture of the stateless blockchain

In account-based stateless blockchains, system states are regarded as key-value pairs and organized in an accumulator (*e.g.*, Merkle tree) where a cryptography *commitment* (*e.g.*, Merkle root) is bound to the current states in the accumulator. Besides, the transaction execution is represented by a transition function $STF(C, B) \rightarrow C'$ which takes in the previous system state commitment C and a batch of transactions B, then outputs the updated commitment C'. Meanwhile, the witnesses W containing the correctness proofs to complete the state transition from C to C' by executing B is generated. Also, a method $VRF(C, B, C', W) \rightarrow \{0, 1\}$ can verify the transition and returns the verification result.

The system consists of on-chain and off-chain parts. As shown in Fig. 1, *proposer nodes* first execute transactions B through $STF()$ to obtain the updated commitment C' based on the full-state maintained in the off-chain storage[2]. Then, they generate the witness/proof W of the transition and provide B, C' and W to *validator nodes*. Meanwhile, the on-chain ledger mainly records changes of the state commitment and is maintained by stateless validators who only store the latest commitment C. Once receiving a transaction batch B with its result C' and proof W, validators perform $VRF()$ against C to verify the transition correctness. If verified, validators record

[1] Most existing works only focus on UTXO-based cryptocurrencies [8, 13, 41].

[2] Some designs [9, 27] also adopt the concept of archive nodes who keep the entire off-chain data to help to generate auxiliary data including the proofs.

C' on-chain through consensus protocols. Meanwhile, validators update their local storage of latest commitment value to C'.

The proof can be achieved by cryptography techniques such as verifiable computing and zero-knowledge proof (*e.g.*, zk-SNARKs [7], example designs are [13, 20]) or hardware-based trusted execution environment (*e.g.*, SGX [14], example designs are [29, 44]). On the other hand, in permissioned chains, the proof can also be predetermined policies such as requiring validators to obtain enough execution results signed by authorized endorsement nodes. Generally, the proof has the feature that it is costly to generate but efficiently to verify a proof (*e.g.*, zk-SNARKs takes hundreds of second to generate and milliseconds to verify a proof). Thus, the processing latency is a notable issue for the stateless chain.

2.2 Execute-order-validate (EOV) Model

Most existing systems (*e.g.*, Bitcoin [34], Ethereum [42], Quorum [12], etc.) adopt the order-execute paradigm where transactions are executed after ordering. Meanwhile, existing stateless designs inherit such a paradigm where either validators order and execute transactions or proposers provide the execution results of transaction batches which are then recorded on-chain by validators.

Hyperledger Fabric [5] introduces a three-phase execute-order-validate (EOV) architecture to support parallel transactions. Specifically, a transaction is first executed by a set of *endorsing peers* appointed by an endorsement policy to obtain its execution result, read/write sets, and endorsement response. Since endorsing peers execute transactions concurrently, it increases the processing speed. When the transaction initiator collects sufficient endorsement, it transmits the transaction with obtained auxiliary data to *ordering services* to determine the total order of transactions. Besides, a consistent view of the order is maintained by leveraging a consensus protocol. Once upon receiving ordered transactions, each peer validates them by checking whether they fulfill the endorsement policy or there exists read/write conflict. Finally, validated transactions take effect on peer's local blockchain states and invalid transactions are aborted and re-invoked by initiators.

In this work, we adopt a similar design pattern to execute and order transactions first and postpone the validation to the last step before committing (details will be introduced in Sec. 3).

2.3 Optimistic Concurrency Control (OCC)

The idea of OCC [26] is to assume the data contention in transactions can barely occur, which has been adopted by many blockchain works in the database area [15, 36, 39]. It allows a transaction to be executed and validated before committing. During validation, it checks if conflicting modifications occur to decide if to commit or abort a transaction. In this work, we adopt a similar idea to deal with the concurrency issue.

3 CAT FRAMEWORK

In this section, we propose the CAT framework and introduce basic roles, data model, transaction processing process, and its lifecycle.

3.1 Overview

This paper aims to propose a stateless blockchain that supports concurrent transaction processing to improve the average transaction processing latency and effective throughput. To be adaptable to existing stateless designs, we inherit three common roles in a system.

Roles. As shown in Fig. 2, CAT has three roles (in blue font).

- **Clients:** initiate transactions and raise requests to the proposer. Besides, they need to prove their transaction initialization validity (*e.g.*, signature) for applications that require authentication.
- **Proposers:** have access to entire **off-chain** storage and execute transactions on behalf of the client. Besides, pack up transactions into batches and provide the execution result and corresponding correctness proofs to the validator.
- **Validators:** have access to a bounded amount of data and are responsible for validating the transaction execution and updating the **on-chain** ledger through consensus protocols.

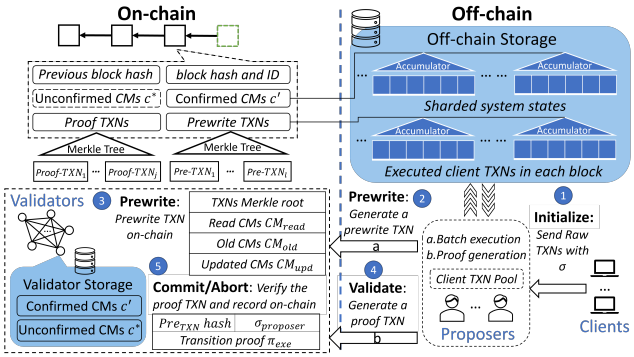


Figure 2: Basic architecture of the CAT framework

To address the challenges listed in Sec. 1, which are concurrent transaction processing, broader usability, and optimization of system parallelism, compared with the existing stateless design, **CAT focuses on the proposer, validator and blockchain ledger structure**, and introduces the following new features:

(1) **Commitment Sharding:** Instead of organizing all system states into one commitment, we shard the states into disjoint subsets with one commitment for each subset to ease the commitment access dependency of different transactions. (Sec. 3.1)
(2) **Parallel Processing:** Without a centralized coordinator, we enable multiple proposers to execute client transactions in batches and propose execution results and proofs simultaneously which further improving the system throughput. (Sec. 3.1)
(3) **Prewrite:** We decompose batch transaction execution and its time-consuming correctness proof generation such that the execution result can be prewritten on-chain first to reflect the commitment update. Thus, the system works in the pipeline, reducing the processing latency. (Sec. 3.2)
(4) **Unconfirmed Commitments:** Validators maintain both the latest confirmed commitments corresponding to committed transactions and unconfirmed commitments corresponding to prewrite transactions. (Sec. 3.3) Commitments are maintained in a novel data structure named commitment forest for detecting and resolving concurrent commitment accessing. (Sec. 4)
(5) **Optimized Batch Selection:** With the commitment forest, the off-chain transaction batch selection at the proposer end is also optimized to maximize the system parallelism. (Sec. 5)

Next we introduce compulsory components and related definitions to achieve the new features in details.

Commitment Sharding. Initially, the state commitment is for integrity checks. For instance, Ethereum uses the Patricia trie [42] to organize the *world state*, and the root is maintained in each block as a snapshot of the system state. However, in a stateless chain, one commitment for all states can easily become a bottleneck. Because the state transition proof is bound to the commitment value and any state change will trigger the commitment update, it forces the transaction processing to be serialized.

Recently, sharding technology has been widely adopted in blockchain systems to enable parallelism [3, 4, 15, 25, 45, 48] where the system states are horizontally partitioned into multiple subsets. As shown in Fig. 2, we adopt a similar idea to shard the states into m disjoint subsets (*e.g.*, according to the state address) and maintain a list of m commitments corresponding to the accumulator of each subset.[3] Besides, we define the state subsets as follows:

Definition 3.1. **State subsets.** Given all system supported state addresses S, we divide it into m disjoint subsets where $S = \bigcup_{i=1}^{m} S_i$ and $\forall i, j\ S_i \cap S_j = \emptyset$. In each subset S_i, the key-value pairs of state addresses and their values are maintained in an accumulator with a cryptography commitment whose index is CM_i and value is $CM_i.v$.

We leave the configuration of m as a system parameter which also introduces a trade-off between parallelism and storage cost. We discuss the details in Sec. 6. Then we introduce the notations of the *client transaction* sent by clients to access system states.

Definition 3.2. **(Client) Transaction.** A transaction sent by a client is represented by a tuple $T_i : (RCM_i, WCM_i, c_i, \sigma_i)$ where RCM_i and WCM_i are the indices of commitments, which the states to be read/written by T_i belong to. c_i is the transaction size and σ_i is the signature of the client. We denote $RW_i = RCM_i \bigcup WCM_i$.

Here, c_i can be T_i's storage cost or consumed gas (*i.e.*, operation cost in the Ethereum), which has a fixed precision. Thus, w.l.o.g., we assume $\forall c_i, c_i \geq 1$ and c_i is an integer. Note that RW_i is a set of indices ($RW_i \subset \{1, 2, ..., m\}$). Since a transaction may read/write multiple states distributed in one or more state subsets, we have $|RW_i| \geq 1$. However, the difference of $|RW_i|$ value does not bring an intrinsic difference to process client transactions. We then overview the basic idea and steps of transaction processing based on Fig. 2.

Parallel Transaction Processing. Inspired by the EOV model, we enable multiple proposers to execute batches of client transactions off-chain in parallel. Different from existing stateless chains, we allow proposers to first record unconfirmed transactions execution results without proofs on-chain by a prewrite transaction.

Definition 3.3. **Prewrite transaction.** Given a client transaction batch T' ordered in a Merkle tree with root T'_{root}, a prewrite transaction is represented by a tuple $B : (T'_{root}, CM_{read}, CM_{upd}, CM_{old})$ where $CM_{read}, CM_{upd}, CM_{old}$ are $< CM_i, CM_i.v >$ pair sets and $\bigcup_{\forall CM_i \in CM_{read}} \{CM_i\} = \bigcup_{\forall T_i \in T'} \{RCM_i\}$, $\bigcup_{\forall CM_j \in CM_{upd}} \{CM_j\} = \bigcup_{\forall CM_j \in CM_{old}} \{CM_j\} = \bigcup_{\forall T_i \in T'} \{WCM_i\}$.

In particular, CM_{read} represents the commitments with values to be read by transaction batch T' and CM_{old} and CM_{upd} represent the commitments with values to be written by T' from CM_{old} to CM_{upd}. By doing so, proposers can be aware of the in-processing updates on each state commitment, thus, adjusting their strategies to avoid generating stale or conflicting proofs and maximize the parallelism brought by the sharding of commitments (detailed in Sec. 5). Later on, proposers generate and send corresponding execution proofs of the unconfirmed transactions in a proof transaction.

Definition 3.4. **Proof transaction.** Given a prewrite transaction B, its proof transaction is represented by a tuple $P_B : (B.hash, \pi_B, \sigma_B)$ where $B.hash$ is the hash of B, π_B is the execution correctness proof of T' in B and σ_B is the signature of proposer.

Once the proof transaction is on-chain, the prewrite transaction is committed. We summarize the processing steps as following:

(1) A client invokes a raw transaction T_i and sends a request with the authentication data (*e.g.*, signature) to proposers. Once receiving T_i, the proposer simulates T_i to obtain RCM_i, WCM_i and c_i, then, add T_i to the local transaction pool.
(2) The proposer select and pre-orders a batch T' of T_i (according to the simulation result. Details are in Sec. 5) and organize T' in a Merkle tree. Then execute T' and send its Merkle root with the aggregated execution result in a **prewrite transaction** B to validators. After that, the proposer starts to generate the required state transition proof π_B.
(3) Validators update affected state commitments based on B, and record the unconfirmed commitment accesses both locally (detailed in Sec. 4.1) and on-chain through a consensus protocol.
(4) Once the prewrite transaction B of T' has been recorded on-chain and the corresponding proof π_B is successfully generated, the proposer sends π_B in a **proof transaction** P_B to validators.
(5) Validators verify π_B and perform concurrency control mechanism (detailed in Sec. 4.2), to make the commit/abort decision on T' and update their local storage. If commit, validators then record the proof transaction P_B on-chain.

On-chain Block Structure. As shown in Fig. 2, an on-chain block has following components: 1) The block ID (height), hash and the previous block hash to form a chain of blocks. 2) Unconfirmed commitment updates in tuples $(CM_i, CM_i.v)$, and prewrite transactions organized in a Merkle tree (to preserve the order[4]), updating the commitments. 3) Confirmed commitment updates with the same format as unconfirmed commitments and the proof transactions organized in a Merkle tree, proving the update correctness. Meanwhile, validators maintain both confirmed and unconfirmed commitments locally, organized in a novel data structure named commitment forest (detailed in Sec. 4).

We then introduce the lifecycle of a transaction in details.

3.2 Initialize and Prewrite

Initialize. A client initializes a transaction T_i defined in Def. 3.2 and sends it to proposers. As shown in the *initialize* phase of Algo. 1, once receiving T_i, a proposer first checks its validity according to σ_i, then, simulates T_i on current system states S to obtain the involved commitment indices RCM_i/WCM_i and the transaction size c_i. The purpose of simulation is for selecting an optimal transaction batch to maximize the system parallelism which will be detailed in Sec. 5. After that, T_i with its simulation result is added into the local transaction pool T. Note that, the transaction pool of each proposer can be different.

[3] In this work, we do NOT specially shard nodes into subsets. Instead, any node in any role has the same view of the state subsets and the corresponding commitment list. Meanwhile, existing works [3, 4, 15, 25, 46, 48], sharding validators to parallel the consensus, are applicable and orthogonal to our work

[4] Note that, a block can contain multiple prewrite and proof transactions.

Prewrite. As shown in the *prewrite* phase of Algo. 1, a proposer periodically selects a client transaction batch T', whose capacity is restricted by a system parameter k, from the pool T. Meanwhile, the proposer determines an internal order of T' and organizes T' into a Merkle tree with root T'_{root} accordingly, then, executes T' based on current system states to obtain accessed commitments (*i.e.*, updated from CM_{old} to CM_{upd} and CM_{read} to read). Then, a corresponding prewrite transaction B is generated and sent to validators who then update their local storage of unconfirmed commitment accesses and record B in an on-chain block through the consensus protocol.

Algorithm 1: Transaction prewrite (at the proposer end)

Initialize Input : Client request (T_i, σ_i), transaction pool T, system states S and commitment list CMs.
Assert $Validate(T_i, \sigma_i)$;
$\langle RCM_i, WCM_i, c_i \rangle \leftarrow Simulate(T_i, S, CMs)$;
$T \bigcup \{T_i : (RCM_i, WCM_i, c_i, \sigma_i)\}$;
Prewrite Input : Transaction pool T, system states S, commitment list CMs and batch capacity k
$T' \leftarrow Selection(T, k)$; // **details are in Sec. 5**
$T'_{root}, CM_{read}, CM_{upd}, CM_{old} \leftarrow Execute(T', S, CMs)$;
$B \leftarrow PrewriteTransaction(T'_{root}, CM_{read}, CM_{upd}, CM_{old})$;
Broadcast B to validators;

Intuitively, prewrite is to *soft-lock* on CM_{old} and CM_{read} and inform other proposers to avoid resources-wasting with stale or conflicting proofs. However, prewrite transactions are uncommitted. Thus, other proposers can choose to 1) follow up unverified results to continue their process; 2) avoid accessing the soft locked commitments; or 3) compete with unconfirmed results to transit commitments to other values. Choice 1 makes the system work in the pipeline, producing the ideal concurrency of the proof generation when there is no failure and malicious behavior, and choice 3 is the worst, providing no concurrency and introducing high abort rate. Besides, choice 2 is the most common case. In fact, the fewer commitments a prewrite transaction soft locks, the higher parallelism the system can have. Thus, to achieve the optimal parallelism, we design the client transaction selection strategy for proposers to minimize the commitment contention, detailed in Sec. 5.

3.3 Validate and Commit/Abort

Validate. After a prewrite transaction B is on-chain, it waits for the execution correctness proof to validate and commit its inside client transaction batch. As shown in Algo. 2, a proof π_B is to prove the following facts: 1) the read sets of T' are correctly obtained (*i.e.*, they are valid members of the underlying accumulators represented by commitments CM_{read}). 2) the state transition in the write sets of T' can correctly update corresponding commitment-value pairs from CM_{old} to CM_{upd}. To complete the proof, several techniques can be applied depending on the chain environment.

For instance, in permissionless chains, verifiable computation (*e.g.*, SNARKs [6]) and zero-knowledge proof (*e.g.*, zk-SNARKs [7]) can be used, meanwhile in permissioned chains, trusted execution environment (*e.g.*, SGX [14]) and predetermined policies are applicable. Since CAT has no restriction on the consensus protocol and proving technique, it can be applied in both environments.

Finally, a proof transaction P_B consisting of B's hash, the proof π_B, and the signature of B's proposer is then sent to validators.

Commit/Abort. As shown in Algo. 2, upon receiving P_B, validators check if the proposer signature matches the proposer of B. Besides, $Expired()$ checks if the on-chain block containing B is more than τ blocks behind the latest on-chain block where τ is a dynamic parameter to ensure the system liveness (detailed in Sec. 6). Next is to check if $\pi_B \in P_B$ can correctly prove the mentioned facts in the validation phase. If not, B is aborted which is to simply ignore its prewrite results. Otherwise, we try to commit B by packing P_B into the upcoming on-chain block as the abort/commit evidence.

Algorithm 2: Transaction validate and commit/abort

Proposer-end : Once observe prewrite transaction B is on-chain.
$T', CM_{old}, CM_{read}, CM_{upd} \leftarrow B$;
$\pi_B \leftarrow Prove(T', CM_{old}, CM_{read}, CM_{upd})$;
$P_B \leftarrow ProofTransaction(B.hash, \pi_B, \sigma_B)$;
Broadcast P_B to validators;
Validator-end : Once receive P_B for an on-chain transaction B
/* **details of Abort() and Commit() are in Sec. 4.2** */
Assert $Validate(P_B, \sigma_B, B)$;
$B, CM_{old}, CM_{read}, CM_{upd}, \pi_B \leftarrow P_B$;
if $Expired(B, \tau)$ **then** *Abort(B)* ;
if not $VRF(B, CM_{old}, CM_{read}, CM_{upd}, \pi_B)$ **then** *Abort(B)* ;
else *Commit(B) and record P_B on-chain by the consensus protocol*;

As CAT allows multiple proposers to concurrently propose prewrite and proof transactions, to abort or commit a prewrite transaction, we need to not only maintain the validators' local storage including both confirmed and unconfirmed commitment values, but also perform the concurrency check. We introduce the details of concurrency control and *Commit()* and *Abort()* functions in next section.

4 COMMITMENT FOREST

While parallel transaction processing of proposers improves the system performance, it also brings the concurrency problem. Thus, in this section, we introduce a novel data structure named commitment forest to maintain the commitment access of general-purpose transaction batches, which assists validators in dealing with the concurrent execution correctness attestation and for proposers to determine their transaction batch selection strategy.

concurrency problem. Since multiple proposers process transactions asynchronously, it is possible to occur the read/write contention on commitment values, affecting the commit/abort of transactions. We use an example to illustrate the problem.

Example 4.1. Suppose there are three prewrite transactions B_1, B_2 and B_3 with hash values 206, 207, and 208. Transaction batch in B_1 read states accumulated in commitment CM_2 with value CM_2^1 and write states to update CM_2^1 to CM_2^2. Thus, it is pending on the proof $\pi_1 : STF(CM_2^1, B_1) \rightarrow CM_2^2$. Meanwhile, B_2 writes the commitments of subset 2 and 3 from CM_2^1 and CM_3^1 to CM_2^3 and CM_3^2, pending on the proof $\pi_2 : STF(\{CM_2^1, CM_3^1\}, B_2) \rightarrow \{CM_2^3, CM_3^2\}$. Finally, B_3 follows up the updates of B_2 to read states accumulated in CM_2^3 and CM_3^2 and write CM_2^3 to CM_2^4, pending on the proof $\pi_3 : STF(\{CM_2^3, CM_3^2\}, B_3) \rightarrow \{CM_2^4, CM_3^2\}$. Note that, π_1 conflicts with π_2, since they involve the same input value but result in different output. Also, the commit of B_3 depends on B_2, since the input of π_3 involves the output of π_2.

The concurrency problem can be viewed from two aspects. For validators, a mechanism is required to determine whether client

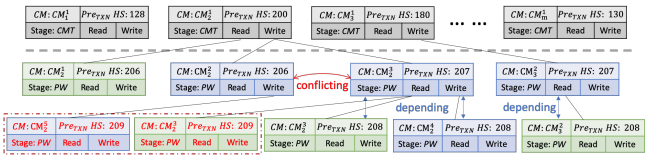


Figure 3: Example structure of the commitment forest

transaction batches represented by prewrite transactions can be committed or aborted (solved in Sec. 4.2). For proposers, it is equally important to select suitable client transactions from the local pool to minimize commitment contention and maximize system parallelism when forming a prewrite transaction (solved in Sec. 5). Note that, to resolve the concurrency problem, it is essential to maintain the accessed commitment values and record the access dependency and conflict. Thus, we design CM-forest to assist the decision making.

4.1 Structure and Construction

Since there are m disjoint subsets, to trace the read/write of each subset's commitment based on the prewrite and proof transactions, we build a forest with m tree roots. Specifically, each tree node contains the following fields: 1) Commitment value (CM) that a client transaction batch in a prewrite transaction reads or writes. 2) Hash (HS) of a prewrite transaction. 3) Stage of the client transaction batch inside a prewrite transaction, including prewrite (PW), pre-commit (PC) and commit (CMT). 4) Read/Write child nodes represent that the prewrite transaction in the child node performs read/write operations on the commitment value recorded in the parent node. Meanwhile, we define each root node as a write node, recording the latest commitment value of each subset, updated by committed client transaction batch. Besides, we use a table E to record the read/write node entries for each prewrite transaction B (*i.e.*, $E[B.hash].read$ and $E[B.hash].write$). When a prewrite transaction is consented by validators and recorded on-chain, one can follow Algo. 3 to insert the prewrite information into the CM-forest.

Algorithm 3: CM-forest insertion for prewrite

Input: Prewrite transaction B, CM-forest roots rts and entries E.

1 $CM_{old}, CM_{read}, CM_{upd} \leftarrow B$;
2 **for** $cm \in CM_{read}$ **do**
3 Find the $node$ where $node.CM = cm$ by BFS/DFS on all "write children" rooted at $rts[cm.index]$;
4 $nd \leftarrow Node(cm, B.hash, "prewrite")$;
5 $node.read.add(nd)$ and $E[B.hash].read.add(nd.addr)$;
6 **for** $i = 0; i < length(CM_{old}); i++$ **do**
7 Find the $node$ where $node.CM = CM_{old}[i]$ by BFS/DFS on all "write children" rooted at $rts[CM_{old}[i].index]$;
8 $nd \leftarrow Node(CM_{upd}[i], B.hash, "prewrite")$;
9 $node.write.add(nd)$ and $E[B.hash].write.add(nd.addr)$;

Based on Example 4.1, suppose three prewrite transactions are recorded on-chain in the order of B_1, B_2 and B_3. By following Algo. 3 we can construct a CM-forest as shown in Fig. 3 (without two nodes in the red box).

4.2 Concurrency Control at the Validator-end

As discussed at the beginning of this section, for validators, a mechanism is required to ensure the committed (client) transaction history is conflict serializable. That is, after receiving a proof transaction of a prewrite transaction, the validators can decide which (client) transactions represented by prewrite transactions need to be committed and aborted with the guarantee of conflict-free.

Recall that in Algo. 2, validators decide to commit/abort a transaction batch by calling *Commit()*/*Abort()* functions. With the construction of CM-forest, we now detail these two functions which need to refer to and operate on the CM-forest.

Algorithm 4: Commit

Input: Prewrite transaction B, CM-forest roots rts and entries E.

1 /* **Pre-commit phase to change the tree node stage** */
2 **for** $nd \in (E[B.hash].read \bigcup E[B.hash].write)$ **do**
3 **Assert** *stage of all nd's sibling write nodes is "prewrite"*
4 *Change the stage of all nodes in E[B.hash] to "pre-commit"*
5 /* **Commit phase to merge tree nodes** */
6 **for** $nd \in (E[B.hash].read \bigcup E[B.hash].write)$ **do**
7 **Assert** *nd.parent.stage == "pre-commit" or "commit"*;
8 **if** *nd is a write node* **then**
9 **Assert** *for all rn of nd's sibling read nodes, either rn.stage == "prewrite" or rn.hash = B.hash*;
10 **for** $nd \in E[B.hash].read$ **do** *delete nd*
11 **for** $nd \in E[B.hash].write$ **do**
12 **for** $cnd \in nd.read$ **do**
13 **if** *cnd.stage == "pre-commit"* **then** *Commit(cnd.preTXN)*
14 **for** $cnd \in nd.write$ **do**
15 **if** *cnd.stage == "pre-commit"* **then** *Commit(cnd.preTXN)*
16 *Abort all siblings of nd and merge nd with nd.parent by replacing CM, read/write fields of nd.parent to corresponding files in nd*
17 **if** $nd.parent \in rts$ **then**
18 *change its hash to the latest merged node hash in the branch*;

The detail of *Commit()* function is shown in Algo. 4. The validator first pre-commits B by checking the stage of all sibling write nodes of $B's$ corresponding tree nodes. For example, in Fig. 3, the sibling write node of B_2 is the write node of B_1 trying to update CM_2 from CM_2^1 to CM_2^2. Thus, only when B_1 is in the prewrite stage, we change the stage of all read/write nodes of B_2 to pre-commit. The pre-commit condition indicates that the required proof of B has been verified. After that, we check if all stages of $B's$ read/write nodes' parents are pre-commit or commit. If so, we then check if for each sibling read node of $B's$ write nodes, either it is in prewrite or it is $B's$ read node. These two conditions (lines 6-9 in Algo. 4) ensure that we can safely merge the pre-commit result of B with its parents. For instance, in Example 4.1, when $B_2's$ proof π_2 is first generated and verified, making B_2 become pre-commit, since its parents (roots of CM_2 and CM_3) are all in commit stage and the sibling read node ($B_1's$ read node) of $B_2's$ write node is in prewrite, B_2 is ready to be committed and its forest branches can be merged.

During the merge phase (lines 10-18 in Algo. 4), we remove $B's$ read nodes and for each $B's$ write node, we first check if any of its child read nodes is in pre-commit. If so, we try to commit it. Similarly, we then check its child write nodes. After that, we abort all siblings of $B's$ write nodes and merge $B's$ write nodes with their parents by replacing values in CM, read and write child nodes fields of parent nodes by the values of $B's$ write nodes. Finally, if the parent is a root node, the node hash is set to the hash of the prewrite transaction, last updating the CM field. After each successful merge, we remove entries of B from the table E.

Follow up the condition shown in Fig. 3, suppose we enter the merge phase of B_2 now, we first check $B_3's$ read nodes then its write node. The semantic meaning is that suppose B_3 is committed before B_2 which is still in prewrite, B_3 cannot be merged with B_2. However, when B_2 is committed, it also triggers the merge of its child B_3.

In addition, *Abort()* is called in these cases: 1) proof validation fails. 2) commit and merge a block, or 3) garbage collection due to the expiration of blocks (detailed in Sec. 6). The main process of *Abort()* is to: 1) remove all $B's$ nodes and successive branches in CM-forest. 2) directly ignore the on-chain prewrite transactions. We refer the details to the appendix.

5 PARALLELISM OPTIMIZATION

As discussed in Sec. 4, another angle to view the concurrency problem is from the proposers who asynchronously select client transactions in different local pools without knowing others' selection strategies. Thus, it is easy to cause the commitment value access contention, causing a high client transaction abort rate. To reduce the abort rate and maximize the system parallelism, a proposer has two aspects to consider:

On the one hand, we should avoid to select transactions that can never be committed. An example is the batch 209 in Fig. 3 (in red font). Whether 209 can be committed depends on two conflicting updates (write nodes of B_1 and B_2). Because, at least one of them will be aborted, it makes 209 have to be aborted. To address this point, a proposer can maintain the CM-forest such that they are aware of if data contention exists. In particular, before forming a prewrite transaction B, a proposer can simulate client transactions based on Algo. 3 and Algo. 4 to verify if B can be successfully committed. Meanwhile, CM-forest can also inform proposers if a prewrite transaction is aborted before they obtain its proof. Thus, it also prevents resource-wasting from generating invalid proofs.

On the other hand, based on the simulation result, a proposer can select a transaction batch with minimized commitments access. In this case, the batch has minimized commit dependency on other batches, which also maximizes the system parallelism. Thus, to address this point, we model and address an optimization problem named *minimum subset cover* (MinSC) in the following subsections.

5.1 Problem Definition

Definition 5.1. **Minimum Subset Cover (MinSC) Problem.** Given state subsets $S = \{S_1, S_2, ..., S_m\}$, n client transactions $T = \{T_1, T_2, ..., T_n\}$ with size c_i for each, and the batch capacity k, MinSC problem is to select $T^* \subseteq T$ to process as many transactions in a batch while minimize covered state subsets count by the union of RW_i where $\forall i, T_i \in T^*$. We formalize the problem as follows:

$$find\ a\ subset\ T^* \subseteq T\ subject\ to \sum_{\forall T_i \in T^*} c_i \leq k$$

$$to\ minimize\ \ m(k - \sum_{\forall T_i \in T^*} c_i) + |\bigcup_{\forall T_i \in T^*} RW_i|$$

Note that, to achieve the optimization goal, the batch space needs to be fully occupied where for the optimal solution T^*, $\nexists T_i \in T - T^*$ such that $c_i \leq k - \sum_{\forall T_j \in T^*} c_j$.

Hardness Analysis. Inspired by the NP-hardness proof of the *set union knapsack problem* [21], we prove the NP-hardness of MinSC by considering its restricted decision version problem named RD-MinSC where $\forall T_i, c_i = 1$ and $|RW_i| = 2$. Due to the space limit, we propose the theorem and refer the detailed proof in the appendix.

Theorem 5.2. *MinSC problem is NP-hard.*

5.2 DP-based Exact Algorithm

We first propose a dynamic programming (DP) based exact algorithm for MinSC problem in small-scale (of proposer's local pool).

Basic Idea. In each stage of the DP, we consider if to allow transactions to access a specific state subset. By tracing the state subsets that have been allowed to access each time when viewing a new state subset, we can determine which transactions can be included in the batch. Thus, we can further compute the space occupied by those transactions and the current objective value.

Algorithm 5: DP-based Algorithm

Input : Candidate client transactions T, batch capacity k, state subsets count m.

Initialize $f(0, \emptyset, c) = \infty, \forall c \in \{1, 2, \ldots, k\}$ and $p(m) = \emptyset$;
for *k'=1; k'<=k; k'++* **do**
  for *j=m; j>=1; j−−* **do**
    $cover_j = f(j-1, p(j) \cup \{j\}, k' - c^*(j, p(j), k', T))$
      $+1 - mc^*(j, p(j), k', T)$;
    $uncover_j = f(j-1, p(j), k')$;
    $f(j, p(j), k') = \min\{cover_j, uncover_j\}$;
return $FindSelection()$;

Algorithm 6: Optimal occupancy function $c^*(\cdot)$

Input : Subset index j, selected subsets $p(j)$, remaining capacity k' and candidate client transactions T.

Initialize an empty list txs and a 2-D array $g(,)$;
for *all T_i in T* **do**
  if $S_j \in RW_i$ and $RW_i \subseteq p(j) \cup \{j\}$ **then** $txs.append(T_i)$;
if *total size of transactions in $txn \leq k'$* **then**
  return total size of transactions in txn;
else
  for *i=1; i<=length(txn); i++* **do**
    for *c=1; c <= k'; c++* **do**
      $g(i, c) = \max\{g(i-1, c), g(i-1, c - c_i) + c_i\}$;
  return $g(k', length(txn))$;

Algorithm Details. As shown in Algo. 5, the DP is conducted through the state subsets from S_m to S_1 in a recursive way. Each time we need to decide if to allow transactions to access a subset S_j or not. Meanwhile, we define $p(j)$ as the indices of subsets S_i $(i > j)$ that has been allowed to access when considering S_j. Then we can define the state function as $f(j, p(j), k')$ representing the optimal objective value when considering subset S_j with the selected subsets $p(j)$ and remaining batch capacity k'. Meanwhile, the state transition function is defined as follows:

$$f(j, p(j), k') = min \begin{cases} f(j-1, p(j), k') \\ f(j-1, p(j) \cup \{j\}, k' - c^*) + 1 - mc^* \end{cases}$$

where c^* is computed by a function $c^*(j, p(j), k', T)$ representing the optimal occupancy of the batch space after allowing to cover the subset S_j by given the selection of $p(j)$, remaining space k' and candidate transactions T. Details are shown in Algo. 6. We first pick

up candidate transactions (lines 1-3) by checking the read/write sets. Then, if their total size is no more than the remaining batch capacity, we select them all. Otherwise, we run another DP to find transactions which can occupy the most of remaining capacity (lines 7-9). Finally, the selected transactions can be reproduced through the function $FindSelection()$ by enumerating the completed record tables of both f and g respectively. We omit the details here.

Complexity Analysis. In Algo. 5, the dynamic programming computes every entries of the state table f, meanwhile $p(j)$ can have 2^m possible states. With n candidate transactions, for each entry, it takes $O(nk)$ to perform the DP in Algo. 6 in the worst case. Thus, the time complexity of Algo. 5 is $O(2^m k^2 mn)$.

5.3 Minimum Density-based Approximation

Due to the high time complexity of the exact algorithm, we then propose an approximation algorithm for the large-scale pool size of a proposer with a performance guarantee.

Algorithm 7: Minimum Density-based (MinD) Algorithm

1 For every $T' \subseteq T$ where $\forall T_i, T_j \in T', RW_i = RW_j$ create new transactions with size $\sum_{\forall T_i \in T'} c_i$ of each to form a set T^g;
2 Initialize $min_c = \infty$ and two empty lists DP_c and DP_s;
3 **while** *$k > 0$ and there exists unchecked transactions in T^g* **do**
4 $\forall T_l^g \in T^g$ compute ρ_l and mark T_l^g as unchecked;
5 **while** *pick unchecked T_l^g with the minimum ρ_l in T^g* **do**
6 **if** $T_l^g.size \leq k$ **then**
7 $T^* \bigcup T_l^g$, $DP_c \bigcup T_l^g$, k-=$T_l^g.size$;
8 $T^g.remove(T_l^g)$ and break the inner while loop;
9 **else**
10 run the same lines 7-9 of Algo. 2 on $DP_c \bigcup T_l^g$;
11 **if** *current remaining batch space < min_c* **then**
12 update min_c and record the selection in DP_s;
13 **if** $min_c == 0$ **then** break all loops;
14 mark T_l^g as checked;
15 **if** k < min_c **then Return** T^*; **else Return** DP_s;

Basic Idea. We directly pick transactions in rounds. As a criteria, we define the **density** of a client transaction T_i as $\rho_i = \frac{|RW_i|'}{c_i}$ representing the ratio of the number of additional subsets that T_i accesses to its size. Specifically, suppose we already pick a transaction batch T', their accessed subsets are $\bigcup_{\forall T_j \in T'} RW_j$. Now, for an unpicked transaction T_i, its density is calculated as $|RW_i|' = |RW_i - \bigcup_{\forall T_j \in T'} RW_j|$. Thus, in each round, we pick the transaction with the minimum density until reach the batch capacity limitation.

Algorithm Details. As shown in Algo. 7, we first group transactions accessing the same commitments to form a new set T^g where $\forall T_l^g \in T^g$, the size of T_l^g is the summation size of grouped client transactions. Then we sort and pick $T_l^g \in T^g$ with the minimum density. In each round, there are two cases: **1)** the size of the picked T_l^g can fit in the remaining batch space. Then, we directly select all client transactions in T_l^g and enter the next round to recompute the density. **2)** transactions in T_l^g cannot entirely fit in the space. Then, we cache T_l^g and perform the same DP as the lines 7-9 of Algo. 6 on the union of cached and already selected transactions (set T^*) and trace the selection that can produce the minimum remaining space. Note that, we do not break the inner loop, instead, we keep on checking T^g in ascending order of the density until finding a T_l^g that can be entirely picked or a selection that can fully occupy the space. Finally, we choose the selection, producing the minimum remaining space (lines 15 and 16).

Performance Analysis. To analyze the performance of Algo. 7, we first define some notations. We denote the optimal selection as OPT, the objective value of a selection T' as $f(T')$ and $C(T') = \sum_{\forall T_i \in T'} c_i$. W.l.o.g, we define that the result of Algo. 7 is obtained in the ascending order of T_i ($i = 1, 2, ..., l$) where $\forall j \leq i$, $\frac{|RW_j|'}{c_j} \leq \frac{|RW_i|'}{c_i}$ and T_{l+1} is the first transaction that cannot be selected due to the capacity constraint. We also define $T^i = \{T_1, T_2, ..., T_i\}$ as the selection after selecting T_i. Then, we have:

Theorem 5.3. *The approximation ratio of Algo. 7 is* $1 + \frac{1}{e}$.

Proof. For $f(T^{i-1}) - f(OPT) = (m - \rho_{opt})C(OPT) - (m - \bar{\rho}_{T^{i-1}})C(T^{i-1})$ where ρ_{opt} and $\bar{\rho}_{T^{i-1}}$ are the average density of optimal solution and T^{i-1} respectively. Since Algo. 7 follow the ascending order of the density to select the transaction, for transactions in the space gap between $C(OPT)$ and $C(T^{i-1})$, the term $m - \rho$ is less than $m - \rho_i$, while $C(OPT) - C(T^{i-1}) \leq k$. Hence:

$$\begin{aligned} f(T^{i-1}) - f(OPT) &\leq (m - \rho_i)k = (m - \frac{|RW_i|'}{c_i})k \\ &\leq mk - \frac{f(T^i) - f(T^{i-1}) + mc_i}{c_i}k \\ f(T^i) &\leq (1 - \frac{c_i}{k})f(T^{i-1}) + \frac{c_i}{k}f(OPT) \qquad (1) \end{aligned}$$

Next we use the induction hypothesis to prove the following inequation holds for any stage $i > 0$:

$$f(T^i) \leq (1 + \prod_{j=1}^{i}(1 - \frac{c_j}{k}))f(OPT) \qquad (2)$$

When $i = 1$, we need to prove $f(c_1) = mk - mc_1 + \rho_1 c_1 \leq (2 - \frac{c_1}{k})f(OPT) = (1 - \frac{c_1}{k})f(OPT) + mk - (m - \rho_{opt})C(OPT)$ which is to prove $(\rho_1 - m)c_1 \leq (1 - \frac{c_1}{k})f(OPT) + (\rho_{opt} - m)C(OPT)$. Since $(1 - \frac{c_1}{k})f(OPT)$ is positive, $\rho_1 \leq \rho_{opt}$ (ρ_1 is the smallest density) and $c_1 \leq C(OPT)$, we have (2) holds when $i = 1$.

If (2) holds for $i-1$, combine with (1), we have $f(T^i) \leq (1 - \frac{c_i}{k})(1 + \prod_{j=1}^{i-1}(1 - \frac{c_j}{k}))f(OPT) + \frac{c_i}{k}f(OPT) \leq (1 + \prod_{j=1}^{i}(1 - \frac{c_j}{k}))f(OPT)$

Thus, (2) holds for any $i > 0$. For the last stage $i = l$ which is the result of Algo. 7, $f(T^l) \leq (1 + \prod_{j=1}^{i}(1 - \frac{c_j}{k}))f(OPT)$. When $\forall i, j, c_i = c_j$ the term gets its maximum value $(1 + (1 - \frac{1}{l})^l)f(OPT) \leq (1 + \frac{1}{e})f(OPT)$ which complete the proof. □

Complexity Analysis. The time complexity of transaction grouping is $O(n^2)$ and the nested loops for density-based picking is $O(n^2)$. Although line 10 may enumerate all transactions in the complexity of $O(n^2)$, it can only be invoked in the last outer iteration. Thus, the time complexity of Algo. 7 is dominated by $O(n^2)$.

6 ANALYSIS OF THE CAT FRAMEWORK

In this section, we analyse the correctness, attack model, liveness and storage cost of the CAT framework.

Correctness. Since client transaction batch in each prewrite transaction is already serialized by proposers, we need to prove that:

Theorem 6.1. *Prewrite transaction histories in CAT generated by using the commitment forest are conflict serializable.*

Proof. Assume histories are not conflict serializable. Then, there exists a cycle of prewrite transactions where two consecutive transactions (*e.g.*, B_1, B_2) have one of following cases on the same state commitment with value CM: 1) CM is updated by B_1 then B_2. 2) CM is updated by B_1 then read by B_2. 3) CM is read by B_1 then updated by B_2. We prove that in all cases, B_1 is always committed before B_2. Because if the cycle exists, due to the transitivity, it can produce the contradiction that B_1 is committed before B_1 (itself).
Case 1. When the proof of B_1 is first validated and consented, its stage enters pre-commit iff all its sibling write nodes are prewrite (line 3 in Algo. 4). Thus, either B_1 is committed before B_2 or both of them are aborted.
Case 2. Similar to case 1, when the write node of B_1 first enters pre-commit, its sibling read nodes where B_2 exists cannot enter the pre-commit stage anymore. Thus, B_1 must be committed before B_2.
Case 3. When the read node B_1 first enters pre-commit, if the write node B_2 never enters pre-commit before B_1 is committed, the proof is complete. Otherwise, when they are both pre-commit, during the commit and merge phase, when trying to commit a write node (*e.g.*, B_2 here) it always checks whether a sibling read node (*e.g.*, B_1) is in pre-commit stage (line 9 in Algo. 4) which will stop the commit. Meanwhile, a read node is always committed and merged before a write node (lines 12-15 in Algo. 4) which completes the proof. □
Attack Model. Any role in CAT may incur malicious behaviors. We next discuss potential attacks and how to resolve them.

Malicious Clients: 1) *invoke unauthorized transactions:* it is prevented by cryptography signatures. 2) *modify the same state with different operations* (*a.k.a.* same-chain replay and double-spending attacks [16]): it can be prevented by assigning and checking a globally accessible nonce for each state as Ethereum [42] does.

Malicious Proposers: 1) *use invalid proof to commit a prewrite transaction:* it is prevented with the help of techniques such as SNARKs and TEE. 2) *only prewrite without proofs:* if so, any operation based on the states pre-written by the malicious proposer is invalided, increasing the abort rate. However, there is no critical security issue such as to subvert states since only validators can finalize transactions on-chain. Once detecting the misbehavior, honest nodes can ignore any prewrite raised by the malicious node. Especially in a permissioned chain, proposers' behaviors are restricted and monitored with specific access control mechanisms.

Besides, in a permissionless chain, this problem is similar to motivating the miners of cryptocurrencies to behave honestly which can be solved by an incentive mechanism. For example, only when a prewrite transaction is successfully committed, its proposer can get some rewards, determined by the number of commitments it soft locks to motivate a proposer further to consider the system parallelism when selecting transaction batches (detailed in Sec. 5). We argue that such mechanism design is orthogonal to our work.

Malicious Validators: When recording prewrite and proof transactions on-chain, validators may have Byzantine faults. However, CAT neither modifies the consensus protocol nor shards validators of the underlying blockchain. Thus, the security guarantee of the original consensus mechanism is preserved.
Liveness and garbage collection. The liveness is ensured by preventing the deadlock that a prewrite transaction is neither committed nor aborted due to failure or malicious behaviors (*i.e.*, a prewrite is proposed and never proved). Thus, as described in Sec. 3, we set the expiration factor τ for each prewrite transaction B. In particular, suppose B is in an on-chain block with height 0, and its proof is in a block with height 10. The proof generation time of B is regarded as 10 block intervals. Besides, similar to controlling the PoW difficulty in Bitcoin [23], we dynamically adjust τ based on proof generation time in past blocks to ensure most proposers can provide the proof within τ block intervals. Besides, τ is used for garbage collection of CM-forest where during each traversal of the forest, we abort the tree nodes whose prewrite transaction has expired.
Storage cost. Comparing with the pure stateless design, CAT introduces additional storage to record the sharded commitments and temporary updates in CM-forest. However, the storage is bounded by two system parameters of subset count m and expiration factor τ. In the worst case, there are $2m\tau$ CM-forest nodes and their entries in the table where each of τ prewrite transactions reads and writes states in m commitments. Suppose the size of pointer and hash addresses of each tree node is 64-bits and the stage occupies 2-bits. Besides, the commitment size depends on its implementation (*e.g.*, 384-bits for the vector commitment [28], 256-bits for Merkle tree/Patricia trie [34, 42]). With the block interval of Ethereum ($\sim$ 10s) and proof generation time of ZCash ($\sim$ 600s), $\tau = 600/10 = 60$. Let $m = 1k$, the storage cost will be no more than $2m\tau(64 * 2 * 2 + 2 + 384)bits \approx 9.18MB$.

7 EXPERIMENTAL RESULTS

In this section, we evaluate CAT by implementing a prototype namely zkCAT based a real stateless blockchain namely zkSync and use the pipeline enable version of zkSync as our baseline.
Pipeline enabled zkSync. zkSync is a well-known Ethereum scaling solution by implementing the zk-Rollup protocol [20] which integrates the concept of stateless blockchain. Specifically, a centralized server coordinates the off-chain proposers to process transactions in batches. It utilizes the zk-SNARKs to achieve the execution correctness proof. For a fair comparison, we enable the pipeline of zkSync where blocks are first proposed on-chain without proofs. A block is counted as committed if its proof is obtained and all its previous blocks are committed.
zkCAT. To apply CAT on zkSync, we mainly did the following modifications: 1) remove the coordinator (server) and enable multiple proposers to process transaction independently. 2) shard the universal state commitment into m commitments for m disjoint state subsets. 3) for each execution proof, the proposer needs to specify which subsets the proof is based on. 4) the commitment forest, as well as the concurrency control algorithms, are maintained at both validator and proposer side. Validators commit a block based on the process introduced in Sec. 3.1 and Sec. 4.2. 5) at the proposer-end, we implement the *MinD* algorithm (Algo. 7) to select transaction batches from the transaction pool of each individual.
Real Dataset. To obtain the real transaction information, we process the Ethereum data provided by Google BigQuery [31]. We first extract the transactions in blocks from number 11M to 12M (generated from Oct-06-2020 to Mar-08-2021). There are two transaction types which are token transfer and contract transactions. For token transfer transactions, the read/write set contains state addresses of the sender and receiver. For contract transaction, it may trigger modifications on arbitrary states. Thus, we investigate the logs generated during the execution of contract transactions to

obtain their read/write sets. Given the settings of total system state subsets count (shown in Table 1), we uniformly assign each state to a subset based on its hexadecimal address (*e.g.*, when $m = 16$, state address staring with "0xa26" is assigned to subset 10). Then, we organize each subset into a Merkle tree (with its root as the commitment). Besides, the gas cost is treated as the transaction size and the block gas limitation is treated as the batch capacity k.

Synthetic Dataset. To capture more cases, we also generate synthetic datasets by varying different parameters respectively with others as default values. Details are shown in Table 1 where default values are in bold. In particular, for each client transaction, to obtain its size and accessed subsets, we use one of the *normal* (**by default**), *uniform* and *exponential* distributions to generate the information respectively. Specifically, we set the expected transaction size as a default constant α (*e.g.*, $\alpha = 100$) and vary its standard deviation σ according to Table 1. Meanwhile, the batch capacity is computed as $\alpha\epsilon$ where ϵ is the expected batch transaction count. For the accessed subset count of each client transaction, we vary both the expectation value μ and standard deviation σ. Beside, to determine a transaction accessing which subsets, we use the Zipfian coefficient θ to control the request skewness where larger θ indicates fewer subsets are more frequently accessed by more transactions.

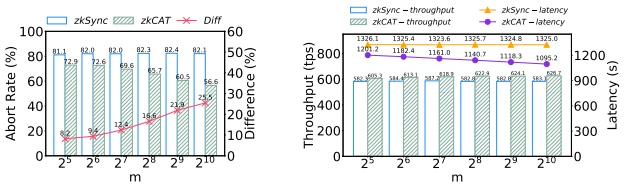
Table 1: Experiment parameter settings

Parameters	Settings
Subset count m	2^2, 2^3, $\mathbf{2^4}$, 2^5, 2^6, 2^7
Candidate client transactions count $\|T\|$	2^7, 2^8, $\mathbf{2^9}$, 2^{10}, 2^{11}, 2^{12}
Expected batch transaction count ϵ	2^4, 2^5, $\mathbf{2^6}$, 2^7, 2^8
Accessed subsets distribution expectation μ	2, **4**, 6, 8, 10
Distribution standard deviation σ	0.1μ, 0.15μ, $\mathbf{0.2\mu}$, 0.25μ, 0.3μ
Zipfian coefficient θ	0, 04, 0.8, **1.2**, 1.6, 2.0

7.1 Performance Evaluation of CAT

Experiment Settings and Metrics. We use both the real Ethereum workloads and synthetic workloads of randomly token transfer between addresses in involved subsets to evaluate CAT. We deploy the systems in the Azure cloud with the Standard E32as_v4 machines [33]. We use 8 clients as consensus nodes which are also the validators of both zkSync and zkCAT, and 8 clients as proposers. We fix the batch capacity of client transactions to 270 TXs/batch to support the aggregation of execution proofs in zkSync and set the τ parameter in zkCAT based on pre-tested proof generation time per block to provide sufficient time for proposers to generate proofs. Besides, we control the on-chain block size of zkSync and zkCAT to contain the same number of batch update transactions in zkSync and prewrite transactions in zkCAT as well as corresponding proofs (w.l.o.g., the size is set to 50 TXs/block by default). For each experiment, we randomly generate 10k transactions and repeat 1k times to compute the average results of the following typical metrics to evaluate the blockchain performance [18]: 1) abort rate; 2) effective throughput; 3) latency; and 4) storage cost of validators.

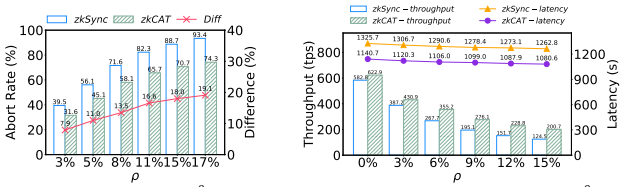
Specifically, transaction abort rate is measured in two cases: 1) proposers incur crash or malicious failure that fails to generate correct proofs for those uncommitted blocks. 2) conflicts caused by commitment access contention of proposers in zkCAT. We capture the former situation by controlling the experimental parameter *failure rate* ρ of each block, and capture the latter by the Zipfian coefficient θ introduced previously. For the effective throughput, it measures how many transactions can be successfully committed in a second, and latency measures the average duration of a transaction from being proposed to committed. For the storage cost, since operations in zkSync including state transition, proof verification, etc. rely on the execution of functions in Ethereum contracts, additional storage is required. For a fair comparison, we only measure the average storage that is compulsory to perform state transition attestation (*i.e.*, the commitment for both methods, the CM-forest and entry tables for zkCAT). Next, we discuss the experimental results and the impact of different parameters.



(a) Abort rate (with ρ=10%). (b) Throughput and Latency (ρ=0%).

Figure 4: Vary the subset count m on the ETH workload.

Impact of the subset count m. As shown in Fig. 4(a), with fixed proof failure rate (10%), the abort rate of zkSync remains high. Because, if a block is aborted due to lack of valid proof, it also invalidates all its subsequent blocks. However, with larger m in zkCAT, the abort rate reduction (diff) compared with zkSync varies from 8.2% to 25.5%. Because more subsets reduce the dependency of accessed commitment values between transactions. Meanwhile, as shown in Fig. 4(b), with no proof failure, the throughput of both methods does not have much difference, since it is determined by the consensus algorithm which we do not change. However, with larger m, zkCAT can reduce at least 9.4% to 17.3% of the processing latency. Because, 1) in the commit phase, with lower dependency, fewer blocks wait for their previous blocks to be committed. 2) with larger m, each subset becomes smaller which also reduces the complexity to generate state transition ZKP.

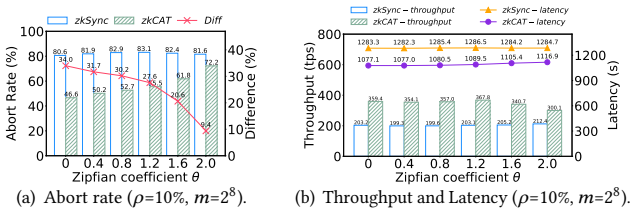


(a) Abort rate (m=2^8). (b) Throughput and Latency (m=2^8).

Figure 5: Vary the proof failure rate ρ on the ETH workload.

Impact of the failure rate ρ. As shown in Fig. 5, with fixed m and increasing ρ, the abort rate difference (diff) between two methods grows from 7.9% to 19.1%. The reason is similar to our previous discussion where zkCAT reduces the commitment access dependency whose benefit is more obvious when ρ grows. Besides, with larger ρ, due to the reduction of abort rate, zkCAT improves the effective throughput from 6.9% to 61.2%, meanwhile, has the average improvement of 14.3% latency due to sharding to 2^8 subsets.

Impact of the Zipfian coefficient θ. As shown in Fig. 6, in synthetic workloads, with larger θ, fewer commitments are more frequently accessed by more transactions. It enlarges the transaction correlation. Thus, the difference of abort rate, the improvement of effective throughput and latency, reduce from 34% to 9.4%, 81% to 41.3% and 16.1% to 13% respectively. It implies that CAT performs



(a) Abort rate (ρ=10%, m=2^8). (b) Throughput and Latency (ρ=10%, m=2^8).

Figure 6: Vary the request skewness Zipfian coefficient θ.

better when the transaction correlation is lower, since the key idea of CAT is to decompose the dependency between irrelevant transactions which cannot ease the intrinsic data contention. Besides, in our Ethereum application workloads evaluation, the situation that transactions have intensive correlation can rarely happen.

Table 2: Storage cost of validators by varying m.

m	2^6	2^7	2^8	2^9	2^{10}
CAT (MKT)	18.43 KB	34.02 KB	61.24 KB	104.12 KB	161.76 KB
CAT (VC)	23.02 KB	42.49 KB	76.50 KB	130.05 KB	202.04 KB
zkSync (MKT)	32 Bytes				
zkSync (VC)	48 Bytes				

Storage cost. The compulsory storage for zkSync is the universal state commitment *i.e.*, the 32 bytes root of the Merkle tree (MKT). To capture more cases, we also calculate the storage after applying vector commitment [28] (VC) which is also widely used to achieve stateless blockchains. Besides, as discussed in Sec. 6, once we set a feasible τ value to provide enough time for proposers to generate the proof, the storage is mainly determined by the subset count m. As shown in Table 2, with larger m, the average storage including the CM-forest and entry table of zkCAT grows linearly from 18.43 KB to 161.76 KB for MKT implementation and 23.02 KB to 202.04 KB for VC implementation. Thus, with notably system performance improvement, CAT does not scarify too much storage space of validators making them substantively stateless.

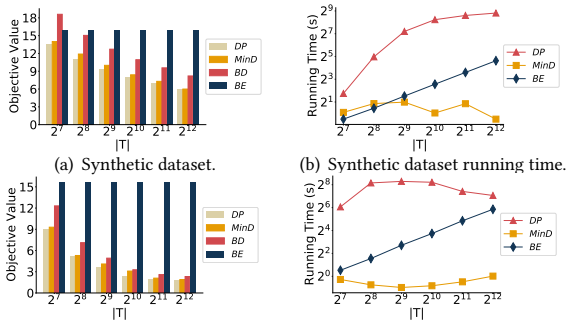
Summary of the CAT evaluation. We conclude that, with a fixed probability of proposer failure and transaction data contention, with more state subset shards, CAT can reduce more abort rate and average processing latency compared with vanilla zkSync. Meanwhile, with a higher proposer failure rate, CAT can reduce more transaction abort rate, resulting in higher effective throughput improvement. Besides, improvements of CAT become more dramatic when transaction correlation becomes lower (with less data contention which is captured by the Zipfian coefficient θ). Finally, CAT can achieve progress with an acceptable cost of validators storage.

7.2 MinSC Algorithms Evaluation

In this subsection, we evaluate the performance of the algorithms proposed in Sec. 5 to show why it is necessary to propose optimization algorithm for proposers to reduce the data contention. Due to the space limit, we only show part of the results and defer more results in the appendix.

Metrics and Implementation. We compare the algorithms proposed in Sec. 5 and use the *best effort* (BE) algorithm as the baseline. Specifically, BE simulates the current transaction packing strategy [47] where miners fully utilize the batch capacity to maximize their profit. Since the profit of each transaction is independent of both its size and content, we uniformly assign a profit value (in the range of (0, 100]) to each transaction and perform the Knapsack dynamic programming algorithm [35]. We mainly evaluate the objective value produced by each algorithm and their running time. Meanwhile, since DP-based algorithm can produce the optimal solution, as proved in Theorem 5.3, we also project the result of $(1+\frac{1}{e})$ times DP-based output (denoted by BD) to see whether the performance guarantee holds. All algorithms are implemented in Python 3.8 and run on a server with 24 cores Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz and 1TB RAM. We output the averaged result by running the algorithms 10 times on the datasets with different random seeds. In the following, we report the algorithm performance by varying parameters shown in Table 1: 1) the candidate client transactions count $|T|$ (for real datasets, we randomly select consecutive transactions with the type in: token transfer, contract, or both). 2) total state subset count m. 3) expected transaction count in a batch ϵ where synthetic datasets use normal distribution.

Impact of the candidate client transactions count $|T|$. As shown in Fig. 7, with the increasing of $|T|$, the objective values produced by DP and $MinD$ decrease. Because with more candidate transactions, there will be more choices to fully utilize the batch space and minimize the modified subsets. However, the result of BE remains the same since it does not take the read/write set into consideration. Comparing with BE, the ratio of $\frac{BE-DP}{BE}$ varies from 0.15 to 0.625 in synthetic and 0.423 to 0.891 in real datasets. While the ratio of $\frac{BE-MinD}{BE}$ varies from 0.125 to 0.625 in synthetic and 0.404 to 0.878 in real datasets. Meanwhile, the result of $MinD$ is always bounded by $1+\frac{1}{e}$ times optimal. As for running time, BE grows linearly with increasing $|T|$ in both datasets, since it performs dynamic programming on every transaction. For $MinD$, it terminates when it finds fully occupied transactions according to the density which is usually determined by the batch capacity. Therefore, in the general case, $MinD$ performs better than the worst-case analysis and takes the least time among the algorithms. While DP takes 2× to 70× more time to find the optimal solution than $MinD$.



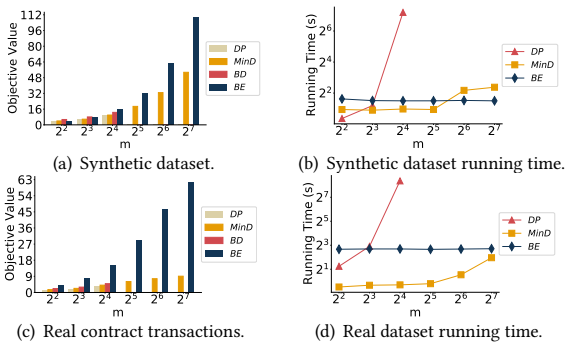
(a) Synthetic dataset. (b) Synthetic dataset running time.

(c) Real contract transactions. (d) Real dataset running time.

Figure 7: Vary candidate transaction count $|T|$

Impact of the total subsets count m. As shown in Fig. 8, with increasing m, the ratio of $\frac{BE-DP}{BE}$ varies from 0 to 0.419 in synthetic and 0.6 to 0.769 in real datasets. While the ratio of $\frac{BE-MinD}{BE}$ varies from 0 to 0.51 in synthetic and 0.6 to 0.852 in real datasets. When m is small (*e.g.*, 4), transactions can easily involve all state subsets. Thus, the results of all algorithms tend to be the same to fully occupy the batch space. When m becomes larger, the DP and $MinD$ algorithms can significantly outperform the baseline which also indicates more subsets can have more parallelism. However, according to the running time, since the complexity of the DP algorithm is exponential to m, when m is larger than 16, the DP algorithm

cannot produce the result in an acceptable time. On the other hand, *MinD* as an approximation algorithm runs as fast as the baseline.

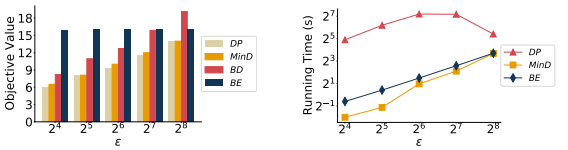


(a) Synthetic dataset. (b) Synthetic dataset running time.

(c) Real contract transactions. (d) Real dataset running time.

Figure 8: Vary total subset count m

Impact of the expected batch transaction count ϵ. With larger batch capacity, the ratio of $\frac{BE-DP}{BE}$ varies from 0.623 to 0.125 and the ratio of $\frac{BE-MinD}{BE}$ varies from 0.591 to 0.125 in the synthetic dataset. Because, by default, there are 2^9 candidate transactions. When ϵ increases to 2^8, half of the transactions will be selected, which causes the objective value to become larger and the difference between *DP*, *MinD*, and *BE* becomes smaller. The running time of *MinD* and *BE* grow linearly with increasing ϵ which meets the analysis. However, for *DP*, since the worst-case only appears when Algo. 6 always enters the else condition, in practice, its running time does not always grow quadratically with the capacity either.



(a) Synthetic dataset. (b) Synthetic dataset running time.

Figure 9: Vary expected batch transaction count ϵ

Summary of the algorithms evaluation. We conclude that both *DP* and *MinD* algorithms can produce better results than the baseline, resulting in less subset accessing contention. Especially when $\frac{k}{|T|}$ is smaller and the average modified subset of transactions is smaller than m. *DP* is more suitable when $m \leq 16$ due to its time cost while *MinD* costs similar to (even less than when $|T|$ increases) the baseline algorithm.

8 RELATED WORKS

Stateless Blockchain. The stateless concept has been successfully applied in permissionless chains [9, 40]. Works such as [8, 13, 22, 41], further reduce the state witness transmission cost with the verifiable computation technique. However, none of the works consider the concurrency problem where the proof of an uncommitted block is obsoleted by another block committed ahead of it.

One closely related work is Aardvark [27] which captures the aforementioned problem by applying the versioning technique. Specifically, apart from storing the system state commitment, validators also keep recently received blocks. Once receiving a stale proof, validators can still verify its correctness and reproduce the latest result by re-executing the block. However, as claimed in [27], the versioning scheme is fairly expressive for simple addition and subtraction operations but limited to undetermined state access which is a normal case for smart contract transactions. Thus, Aardvark cannot support concurrent execution attestation for general-purpose transactions which we are targeting. What's more, the versioning scheme sacrifices too much storage for concurrency. As studied in [27], validators still need to keep a few gigabytes of data. On the other hand, as demonstrated in Sec. 7, with the support of concurrency, validators in CAT only need to keep a few kilobytes of data which makes them remain stateless.

Concurrency Control in Blockchains. Fabric [5] introduces the EOV model enabling the concurrent transaction execution before ordering. Following works such as Fabric++ [39], Fabric# and FastFabric# [36] aim to further reduce the transaction abort rate caused by concurrent execution in different granularity. While, OXII [2] considers this problem in the order-execute model. Besides, some works consider the concurrency problem during the consensus process to record transactions on-chain. Examples are Fabric [5] and AHL [15] where system states are sharded into isolated shards. Transactions within each shard are processed in parallel and concurrent proposed cross-shard transactions are handled by a dedicated node community. On the other hand, CAPPER [1], SharPer [3], Qanaat [4] and PAS [46] deal with the concurrency problem of cross-shard transactions by specially designed consensus protocols. Different from concurrent transaction execution works where a reordering method can reduce the abort rate, **the main concurrency issues in a stateless design that we addressed are:** 1) the abort rate is affected by not only data contention but also the reliability of proposers to provide valid transition proofs. 2) since the proof is bound to the commitment and validators are stateless, reordering cannot be realized by only keeping the commitments. Besides, concurrent consensus works are orthogonal to ours, since we do not modify the consensus protocol used by validators.

9 CONCLUSION

In this paper, we propose CAT, a novel stateless blockchain framework to support concurrent system state commitment update and transaction execution correctness attestation. Specifically, we perform sharding on the commitment and adopt the execute-order-validate model to prewrite the execution result on-chain then do the validation. To deal with the concurrent prewrite, we achieve optimistic concurrency control by letting validators maintain prewrite accesses in a novel data structure named commitment forest. We also model an optimization problem called MinSC, aiming to maximize the system parallelism through transaction selection strategy and prove its NP-hardness. We propose an exact solution and a constant-factor approximation algorithm to address it on small and large scales. We apply CAT on a well-known stateless blockchain named zkSync. Extensive experiments show that CAT can improve 61.2% effective throughput and 17.3% latency on average compared to the original design of zkSync.

REFERENCES

[1] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1385–1398.

[2] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1337–1347.

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*. 76–88.
[4] Mohammad Javad Amiri, Boon Thau Loo, Divyakant Agrawal, and Amr El Abbadi. 2021. Qanaat: A Scalable Multi-Enterprise Permissioned Blockchain System with Confidentiality Guarantees. *arXiv preprint arXiv:2107.10836* (2021).
[5] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 781–796.
[7] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 326–349.
[8] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*. Springer, 561–586.
[9] Vitalik Buterin. 2017. The stateless client concept. https://ethresear.ch/t/the-stateless-client-concept/172.
[10] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual international cryptology conference*. Springer, 61–76.
[11] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.
[12] J. P. Morgan Chase. 2018. A Permissioned Implementation of Ethereum. https://consensys.net/quorum.
[13] Alexander Chepurnoy, Charalampos Papamanthou, and Yupeng Zhang. 2018. Edrax: A Cryptocurrency with Stateless Transaction Validation. *IACR Cryptol. ePrint Arch.* 2018 (2018), 968.
[14] Victor Costan and Srinivas Devadas. 2016. Intel sgx explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.
[15] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.
[16] Dipankar Dasgupta, John M Shrein, and Kishor Datta Gupta. 2019. A survey of blockchain from security perspective. *Journal of Banking and Financial Technology* 3, 1 (2019), 1–17.
[17] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. 2018. Untangling blockchain: A data processing view of blockchain systems. *IEEE transactions on knowledge and data engineering* 30, 7 (2018), 1366–1385.
[18] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1085–1100.
[19] Nelly Fazio and Antonio Nicolosi. [n.d.]. Cryptographic accumulators: Definitions, constructions and applications. ([n. d.]).
[20] Alex Gluchowski. 2019. Zk rollup: scaling with zero-knowledge proofs. https://pandax-statics.oss-cn-shenzhen.aliyuncs.com/statics/1221233526992813.pdf.
[21] Olivier Goldschmidt, David Nehme, and Gang Yu. 1994. Note: On the set-union knapsack problem. *Naval Research Logistics (NRL)* 41, 6 (1994), 833–842.
[22] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. 2020. Pointproofs: Aggregating proofs for multiple vector commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2007–2023.
[23] Reiki Kanda and Kazuyuki Shudo. 2020. Block interval adjustment toward fair proof-of-work blockchains. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 1–6.
[24] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.
[25] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 583–598.
[26] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.
[27] Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, and Nickolai Zeldovich. 2020. Aardvark: A Concurrent Authenticated Dictionary with Short Proofs. *IACR Cryptol. ePrint Arch.* 2020 (2020), 975.
[28] Benoît Libert and Moti Yung. 2010. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *Theory of Cryptography Conference*. Springer, 499–517.
[29] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: a secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 63–79.
[30] matter labs. 2019. zkSync: scaling and privacy engine for Ethereum. https://zksync.io.
[31] Evgeny Medvedev and Allen Day. 2018. Google BigQuery Ethereum Dataset. https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-how-we-built-dataset.
[32] Ralph C Merkle. 1987. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*. Springer, 369–378.
[33] Microsoft. [n.d.]. Azure Virtual Machines. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-memory.
[34] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.
[35] David Pisinger. 1995. Algorithms for knapsack problems. (1995).
[36] Pingcheng Ruan, Dumitrel Loghin, Quang-Trung Ta, Meihui Zhang, Gang Chen, and Beng Chin Ooi. 2020. A transactional perspective on execute-order-validate blockchains. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 543–557.
[37] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 1. IEEE, 57–64.
[38] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 459–474.
[39] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens Dittrich. 2019. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*. 105–122.
[40] Peter Todd. 2016. Making UTXO set growth irrelevant with low-latency delayed TXO commitments. https://petertodd.org/2016/delayed-txo-commitments.
[41] Alin Tomescu, Ittai Abraham, Vitalik Buterin, Justin Drake, Dankrad Feist, and Dmitry Khovratovich. 2020. Aggregatable subvector commitments for stateless cryptocurrencies. In *International Conference on Security and Cryptography for Networks*. Springer, 45–64.
[42] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[43] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*. 141–158.
[44] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2314–2326.
[45] Zihuan Xu and Lei Chen. 2021. DIV: Resolving the Dynamic Issues of Zero-knowledge Set Membership Proof in the Blockchain. In *Proceedings of the 2021 International Conference on Management of Data*. 2036–2048.
[46] Zihuan Xu, Siyuan Han, and Lei Chen. 2021. PAS: Enable Partial Consensus in the Blockchain. In *International Conference on Database Systems for Advanced Applications*. Springer, 375–392.
[47] Andrew Chi-Chih Yao. 2018. An incentive analysis of some Bitcoin fee designs. *arXiv preprint arXiv:1811.02351* (2018).
[48] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 931–948.

A ABORT A PREWRITE TRANSACTION

The meaning of Abort can be viewed from two perspectives: 1) to ignore the effect of a prewrite transaction. Since when the Abort function is called, the prewrite transaction already enters the validate phase and is recorded on-chain. Due to the immutability feature, it is impossible to remove the prewrite transaction from the blockchain. Thus, without further processing and corresponding proof transaction on-chain, the invalid prewrite transaction takes no effect on the system states. 2) to remove the corresponding nodes and entries from the commitment forest. To achieve this, we call the Algo. 8. In particular, we remove all read nodes of a prewrite transaction B. For $B's$ write nodes, not only do we need to remove the node itself, but also we remove all its child read and write nodes which is a recursive calling on the Abort function. Because the failure of B invalidates all prewrite transactions that depend on B.

Algorithm 8: Abort

Input: Prewrite transaction B, CM-forest roots rts and entries E.

```
for nd ∈ E[B.hash].read do
    Remove nd and its entry record in E;
for nd ∈ E[B.hash].write do
    Remove nd and its entry record in E;
    for cnd ∈ (nd.read ⋃ nd.write) do
        Add the prewrite transaction whose hash is cnd.hash into
          the set ToAbort;
while ToAbort != ∅ do
    B' ← ToAbort.pop();
    Abort(B');
```

B HARDNESS PROOF OF THE MINSC PROBLEM

Definition B.1. **Restricted Decision problem of Minimum Subset Cover (RD-MinSC).** Given state subsets $S = \{S_1, S_2, ..., S_m\}$, n client transactions $T = \{T_1, T_2, ..., T_n\}$ with size $c_i = 1$ and $|RW_i| = 2$ for each, the batch capacity k, and a parameter o, RD-MinSC is to decide if there exists a batch $T^* \subseteq T$ where $\sum_{\forall T_i \in T^*} c_i \leq k$ such that the objective value of $m(k - \sum_{\forall T_i \in T^*} c_i) + |\bigcup_{\forall T_i \in T^*} RW_i|$ equals to o.

Here is an example of the RD-MinSC problem.

Example B.2. Given $|S| = 5, |T| = 5, k = 3, o = 3$ and $RW_1 = \{S_1, S_2\}, RW_2 = \{S_1, S_3\}, RW_3 = \{S_2, S_3\}, RW_4 = \{S_3, S_4\}, RW_5 = \{S_4, S_5\}$, to decide if there exists a subset $T^* \subseteq T$ where $|T^*| \leq k = 3$ such that $5(3 - |T_i|) + |\bigcup_{\forall T_i \in T^*} RW_i| \leq o = 3$ can hold.

In fact, given an instance $\mathcal{I}$ of RD-MinSC, we can construct an equivalent problem on a graph $G = (V, E)$ by mapping $S_i \in S$ in $\mathcal{I}$ to vertices $v_i \in V$ in G and $T_j \in T$ in $\mathcal{I}$ to edges $e_j \in E$ in G according to RW_j of each T_j where if $RW_j = \{S_k, S_l\}$, an edge $e_j = (v_k, v_l)$ in G is constructed. The problem is to decide, in G, if there exists a subset $E' \subseteq E$ where $|E'| \leq k \in \mathcal{I}$ such that $|V|(k - |E'|) + |\bigcup_{\forall e_j \in E'} \{v_k, v_l | e_j = (v_k, v_l)\}| \leq o \in \mathcal{I}$.

As shown in Fig. 10(a), it is a graph problem conversion of Example B.2 and there exists a feasible selection $E' = \{T_1, T_2, T_3\}$ such that the objective value is 3 ($|\{S_1, S_2, S_3\}|$) which is also the corresponding solution to the RD-MinSC problem in Example B.2. With such a conversion, we can prove that RD-MinSC problem is NPC by performing reduction from the clique decision problem [24].

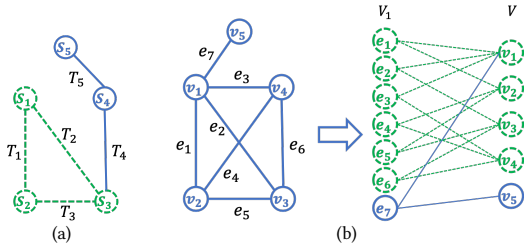


Figure 10: (a) an example of converted RD-MinSC instance. (b) an example to convert a graph to an RD-MinSC instance.

Theorem B.3. *RD-MinSC problem is NP-complete.*

Proof. Consider an instance $\mathcal{J}$ of the clique decision problem [24] where a graph $G = (V, E)$ and an integer α are given to decide if there exists a α-*clique*. We can convert $\mathcal{J}$ to the graph version of RD-MinSC instance $\mathcal{K}$ by doing the following map: 1. construct a vertex set V_1 where $\forall v_i \in V_1, \exists e_i \in E$ of $\mathcal{J}$ (each vertex in V_1 represents an edge in $\mathcal{J}$). 2. define a bipartite graph $B = (V_1 \bigcup V, E^*)$ where $V \in \mathcal{J}$ and an edge $e_j = (v_j^1, v_j^2) \in E^* (v_j^1 \in V_1, v_j^2 \in V)$ means that an edge $e \in \mathcal{J}$ represented by $v_j^1 \in V_1$ is incident to the vertex $v_j^2 \in V$. 3. Let $G \in \mathcal{K}$ be B, $k \in \mathcal{K}$ be $2\binom{\alpha}{2}$ and $o \in \mathcal{K}$ be $\binom{\alpha}{2} + \alpha$. An example graph conversion is shown in Fig. 10(b). Next we prove $\mathcal{J}$ has a feasible solution iff $\mathcal{K}$ has a feasible solution:

Sufficiency: if there is a α-*clique* in $\mathcal{J}$, there must be $\binom{\alpha}{2}$ vertices in $V_1 \in \mathcal{K}$ representing the clique edges, α vertices in $V \in \mathcal{K}$, and corresponding $2\binom{\alpha}{2}$ edges in $E^* \in \mathcal{K}$. Thus, we select these edges in E^* whose object value is $\binom{\alpha}{2} + \alpha$ to get a feasible solution of $\mathcal{K}$.

Necessity: Suppose a feasible subset $E' \subseteq E^* \in \mathcal{K}$ can produce the objective value of $\mathcal{K}$ as $\binom{\alpha}{2} + \alpha$. We define the subgraph $B' \subseteq B$ consisting E' and their endpoints. Meanwhile, the term $|V|(k - |E'|)$ in the objective function must be 0. Meanwhile, since the degree of vertices in V_1 must be 2, for edges in E', they must incident to $\binom{\alpha}{2} + p$ ($p \geq 0$) vertices in V_1 and $\alpha - p$ vertices in V. Since in $G \in \mathcal{J}$ only when an edge e (represented by $e' \in E'$) and its two endpoints $v^1, v^2 \in V$ are all included in $B' \in \mathcal{K}$, the degree of $e' \in B'$ is 2. Thus, there are $\binom{\alpha-p}{2}$ such vertices with degree 2 in V_1 and $\binom{\alpha}{2} + p - \binom{\alpha-p}{2}$ vertices in V_1 with degree 1. Thus, $|E'| = 2\binom{\alpha-p}{2} + \binom{\alpha}{2} + p - \binom{\alpha-p}{2} = 2\binom{\alpha}{2} + \frac{p^2+3p-2\alpha p}{2}$. Since $|E'| \leq 2\binom{\alpha}{2} \Rightarrow p = 0$. Thus, $\mathcal{K}$'s solution must have $\binom{\alpha}{2}$ and α vertices in V_1 and V which also represent the edges and vertices of the α-*clique* in $\mathcal{J}$ which completes the reduction. □

Theorem B.4. *MinSC problem is NP-hard.*

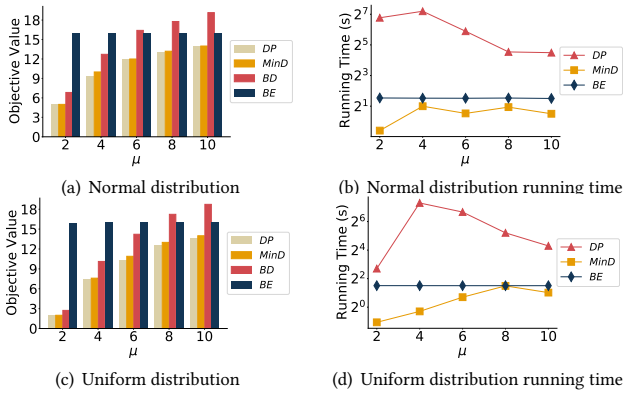
Proof. According to Theorem B.3, we know the optimization version of RD-MinSC is NP-hard, which is also a special case of MinSC problem. Thus, MinSC problem is NP-hard as well. □

C SUPPLEMENTARY EXPERIMENTAL RESULTS

C.1 MinSC Algorithms Evaluation – Impact of the Accessed Subsets Distribution Expectation μ.

With larger μ, a transaction is expected to modify more subsets which make the objective value become larger and the results of *DP* and *MinD* closer to *BE*. For simplicity, we call the synthetic data with normal and uniform distributions of the modified subsets for each transaction as normal and uniform datasets respectively. The ratio of $\frac{BE-DP}{BE}$ varies from 0.688 to 0.125 in normal and 0.874 to 0.144 in uniform dataset. While, the ratio of $\frac{BE-MinD}{BE}$ varies from 0.688 to 0.125 in normal and 0.874 to 0.125 in uniform dataset. It implies that the uniform distribution is more sensitive to μ. Because it has larger variance between the modified subset count of each transaction. Thus, *DP* and *MinD* algorithms have more chances to select transactions with less modified subsets.

While, the running time of *BE* and *MinD* remain steady and *DP* first increases until $\mu = 4$ then go down in both datasets. Because,



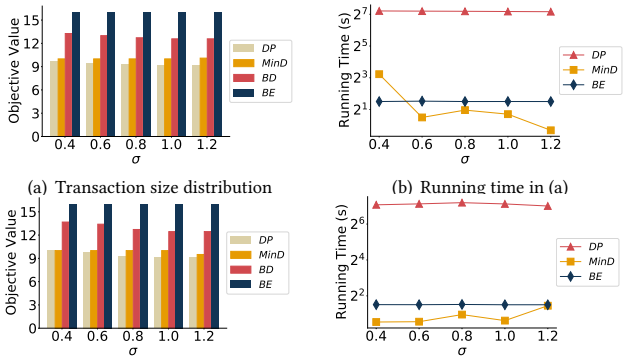
(a) Normal distribution (b) Normal distribution running time

(c) Uniform distribution (d) Uniform distribution running time

Figure 11: Vary expectation μ of accessed subsets count dist.

for DP, larger μ first enlarges the subsets overlap among transactions. Thus, for each state of $p(j)$ in Algo. 5, more transactions are considered. However, since it is more likely that only when $|p(j)| \geq \mu$, there exists transactions to be considered. It makes the number of possible states of $p(j)$ that are necessary to test become smaller which reduces the major cost of DP.

C.2 MinSC Algorithms Evaluation – Impact of the Distribution Standard Deviation σ.



(a) Transaction size distribution (b) Running time in (a)

(c) Accessed subsets distribution (d) Running time in (c)

Figure 12: Vary standard deviation σ of distributions

As shown in Fig. 12, with an increasing σ of the normal distributions in synthetic data, for the transaction size distribution, the ratio of $\frac{BE-DP}{BE}$ and $\frac{BE-MinD}{BE}$ vary from 0.394 to 0.425 and 0.375 to 0.369 respectively. Because, with larger transactions size variance, the density computation of $MinD$ is more likely to be dominated by the size. Thus, after fulfilling the batch space, the modified subsets contributing to the objective goal may not be the optimal. While, for the modified subset count distribution, the ratio of $\frac{BE-DP}{BE}$ and $\frac{BE-MinD}{BE}$ vary from 0.375 to 0.431 and 0.375 to 0.406 respectively. In this case, modified subsets contribute more to the density which make the final result closer to the optimal result.