L2chain: Towards High-performance, Confidential and Secure Layer-2 Blockchain Solution for Decentralized Applications

Zihuan Xu
The Hong Kong University of Science and Technology
Hong Kong SAR, China
zxuav@cse.ust.hk

Lei Chen
The Hong Kong University of Science and Technology
Hong Kong SAR, China
leichen@cse.ust.hk

ABSTRACT

With the rapid development of blockchain, the concept of decentralized applications (DApps), built upon smart contracts, has attracted much attention in academia and industry. However, significant issues *w.r.t.* system throughput, transaction confidentiality, and the security guarantee of the DApp transaction execution and order correctness hinder the border adoption of blockchain DApps.

To address these issues, we propose L2chain, a novel blockchain framework aiming to scale the system through a layer-2 network where DApps process transactions in the layer-2 network and only the system state digest, acting as the state integrity proof, is maintained on-chain. To achieve high performance, we introduce the split-execute-merge (SEM) transaction processing workflow with the help of the RSA accumulator, allowing DApps to lock and update a part of the state digest in parallel. We also design a witness cache mechanism for DApp executors to reduce the transaction processing latency. To fulfill confidentiality, we leverage the trusted execution environment (TEE) for DApps to execute encrypted transactions off-chain. To ensure transaction execution and order correctness, we propose a two-step execution process for DApps to prevent attacks (*i.e.*, rollback attacks) from subverting the state transition. Extensive experiments have demonstrated that L2chain can achieve 1.5X to 42.2X and 7.1X to 8.9X throughput improvements in permissioned and permissionless settings respectively.

PVLDB Reference Format:
Zihuan Xu and Lei Chen. L2chain: Towards High-performance, Confidential and Secure Layer-2 Blockchain Solution for Decentralized Applications. PVLDB, 16(4): 986 - 999, 2022.
doi:10.14778/3574245.3574278

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xzhflying/L2chain.

1 INTRODUCTION

Blockchain is an append-only structure of linked blocks, containing transactions issued by participants who do not trust each other. Consensus protocols ensure the security and reliability of the global view of the on-chain data. With the rapid development, it has evolved from original UTXO-based model which only supports cryptocurrencies (*e.g.*, Bitcoin [49]) to account-based model, supporting Turing-complete transactions with *smart contracts* (*e.g.*, Ethereum [61]). In particular, smart contracts can represent service level agreements (SLAs) among participants, empowering the *decentralized applications* (DApps) where data and SLAs are recorded in contracts without a centralized service provider. It has been used in many scenarios such as finance [59], healthcare [47], crowdsourcing [32, 41], data sharing [26], federated learning [43], etc..

However, low throughput and scalability issues hinder the development of blockchain-based DApps where the consensus protocol is the main bottleneck [23]. For instance, *proof-of-work* (PoW) [49], as a widely used protocol in *permissionless* chains where any node can freely join or leave, can only commit tens of transactions per second (tps), while PBFT [18] in *permissioned* chains with known node identities can achieve hundreds of tps [24]. In addition, privacy is another not well-addressed issue. Specifically, DApps can work individually or collaborate to complete one workflow and some DApps may use confidential logic and data to process transactions. Meanwhile, DApps users may care about their privacy, requiring executing transactions without exposing the details to DApp nodes. What's more, it is essential to ensure transactions are successfully executed in the correct order with the resistance to malicious behavior trying to subvert the system security. Thus, in summary, we need to fulfill three requirements of blockchain-based DApps:

(1) **High Performance.** The transaction processing should have high performance in terms of high throughput and low latency.
(2) **High Confidentiality.** Both intra- and inter- DApp transactions can be processed without exposing detailed information.
(3) **High Security.** Transactions processed by each DApp should be *correctly executed* in an *correct order* to obtain the final result.

Recently, the concept of *layer-2* (L2) solution, building upon a *layer-1* (L1) blockchain, has been proposed to improve the DApps' performance. It is regarded as an orthogonal solution to enhance the consensus protocols of blockchains [9]. Specifically, L2 solution enables DApps to parallelly execute transactions *off-chain* through authenticated and private communication channels. Additionally, it only finalizes the execution result of a transaction batch on-chain once, without recording every single transaction. Thus, compared with a pure L1 chain, L2 solution performs much fewer on-chain consensus, dramatically improving throughput and scalability.

However, existing L2 solutions are still immature and fail to address aforementioned three requirements simultaneously. In particular, existing solutions can be categorized into two types. One is to rely on decentralized *channels* where participants first lock the on-chain collateral and establish private communication channels to exchange authenticated state transitions off-chain. After

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574278

the closure of channels, the final collateral state is recorded on-chain. Examples are *payment channels* for efficient token transfer in cryptocurrencies [22, 33, 50, 55] and *state channels*, supporting general-purpose state transitions [25, 48]. However, since each channel solely processes transactions without the guarantee of L1 consensus protocols, it is possible that malicious nodes can subvert the predetermined execution order, resulting in incorrect system states which fail to fulfill the security requirement [30].

The other is to rely on a centralized but untrusted *executor*. Example works are *commit-chains* [37, 54] where an executor collects and executes transactions off-chain and commits the updated states periodically as checkpoints to an L1 on-chain smart contract, which allows users to verify the transaction execution correctness and challenge the misbehavior of the executor. Besides, with the help of verifiable computation and zero-knowledge proof (ZKP), *rollup protocols* [16, 27, 28] improve the commit-chains by recording the verifiable execution correctness proof of L2 transactions on-chain, ensuring the data availability and reducing the need for users to monitor the executors' behavior. However, centralized processing exposes the transaction details to an untrusted third party, failing to comply with the confidential requirement. Meanwhile, due to the contention of different DApp executors on the L1 state digest and the high computational cost of the ZKP, to finalize the processing result of L2 transactions on the L1 chain, it is restricted to be serialized with high latency (*e.g.*, tens of minutes in zkSync [45]).

To achieve high performance, confidentiality, and security simultaneously, in this paper, we propose a novel layer-2 framework named L2chain. Specifically, **to achieve high performance**, we introduce split-execute-merge (SEM), a novel transaction processing workflow to enable the parallel processing of L2 transactions and we use the RSA accumulator [17, 40] to organize system states, enabling flexible state sharing among DApps. Given a transaction batch, L2 DApp executors first split the global state RSA accumulator to obtain the digest of states appearing in the transaction batch's read/write sets. Then execute transactions to update the partial digest. Finally, L1 validators merge the updated digest back to the global RSA accumulator. Besides, we design a cache mechanism and related optimizations for DApp executors to reduce the transaction processing latency. **To achieve high confidentiality**, we leverage the trusted execution environment (TEE) [57] for DApp executors to execute encrypted user transactions off-chain. Meanwhile, we only maintain the system state digest on the L1 chain without explicitly recording any L2 transaction information. **To achieve high security**, we introduce the two-step execution process at the DApp executor-side. Executors first simulate the transaction batch to obtain the read/write sets and use TEEs to verify and sign on the transaction order determined in the L2 network by DApp specified consensus protocols. Once such information is finalized on the L1 chain, executors then execute transactions based on the finalized order, ensuring the order correctness to prevent the rollback attack [14], violating the determined execution order.

We summarize our contributions as follows:

(1) We present L2chain, a novel layer-2 blockchain framework, offering high performance with flexible state sharing among DApps, transaction confidentiality with stricter privacy and security guarantee to prevent the rollback attack. To the best of our knowledge, this is the first work of its kind.
(2) We propose split-execute-merge, a novel transaction processing workflow, and two-step (TEE) execution at the DApp executor-side. It ensures both transaction confidentiality and execution correctness. Meanwhile, it can withstand the rollback attack.
(3) We introduce the cache witness mechanism with optimizations, assisting layer-2 DApp executors in using the RSA accumulator in transaction processing in L2chain efficiently.
(4) We implement a prototype and conduct extensive experiments to evaluate the performance of L2chain. Results show that L2chain can improve the throughput by 7.1X to 8.9X in a permissionless setting and 1.5X to 42.2X in a permissioned setting.

The rest of paper is organized as follows. We motivate our work and introduce related concepts and works in Sec. 2. Sec. 3 overviews L2chain. Sec. 4 presents state organization and transaction processing in L2chain. Sec. 5 introduces the RSA witness cache mechanism with optimizations. Sec. 6 details the transaction execution steps. We evaluate L2chain in Sec. 7 and conclude our paper in Sec. 8.

2 BACKGROUND AND RELATED WORKS

In this section, we first provide a supply-chain example to motivate our work. Then we introduce main concepts related to our solutions. Finally, we discuss the novelty against related works.

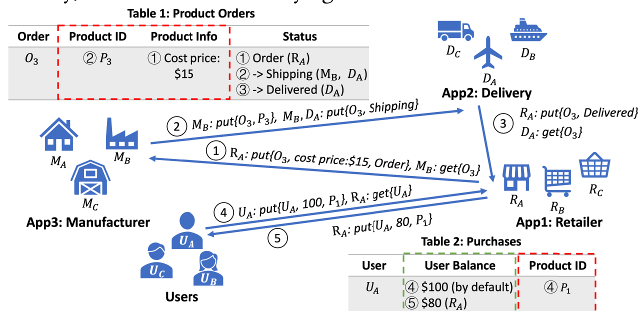


Figure 1: Example in a supply chain scenario

2.1 Motivation Example

Example 2.1. In Fig. 1, there are three DApps: retailer, delivery and manufacturer. Each has three nodes (executors) to provide corresponding services. In this example, there are two workflows:
Workflow 1: All DApps jointly maintain Table 1, recording the product order status. Suppose ***inter-app*** transactions ①-③ achieve the business logic that ①: retailer R_A places an order O_3 with the cost price of $15 and sets the status as "Order"; ②: after getting the O_3, manufacturer M_B puts the product P_3 into this order and assigns delivery D_A to ship the product; ③: once receiving the product, R_A modifies the status of O_3 to "Delivered";
Workflow 2: The retailer DApp maintains Table 2, recording the product purchase information of users. Suppose ***intra-app*** transactions ④ and ⑤ achieve the business logic that ④: user U_A sends her/his current balance and buys the product P_1 in $20; ⑤: after verifying that U_A's balance is sufficient, retailer R_A updates U_A's balance and records the balance state on-chain.

Suppose we use blockchain to maintain the data without a third-party, the three requirements mentioned in Sec. 1 become essential:
High performance: Since workflows 1 and 2 access different states (order O_3 and user balance U_A) without any dependency, transactions can be processed in parallel to improve the performance.

High confidentiality: For each DApp, some data can only be visible internally and only record the data hash on-chain for integrity check. Examples are product cost price in Table 1 and user purchased product in Table 2. For DApp users, some transactions content needs to be even hidden from DApp executors. Such as a user's current balance required in transaction ④ and ⑤.

High security: Two criteria of the transaction need to be ensured:

Execution correctness needs to be publicly verifiable. For example, without exposing user's balance, DApps need to ensure that U_A has sufficient balance to purchase P_1 and U_A's balance is correctly deducted after executing transaction ⑤.

Order correctness to enforce DApps can only execute transactions in a determined order to prevent rollback attacks where the malicious obtains valid but outdated states to 1) cause state inconsistency between L1 and L2, affecting the system security. 2) rollback to outdated states to detect confidential transaction content.

Intuitively, we can encrypt states and transactions and let DApp executors decrypt the inputs and execute transactions in a **black box** that outputs updated encrypted states. Such that executors cannot learn anything from the inputs and outputs. However, DApps' marginal conditions (*e.g.*, indicating invalid inputs) can leak side information. Specifically, a rollback attacker can solely execute uncommitted self-made transactions together with user transactions to trigger marginal conditions, breaking privacy. Here we illustrate how a rollback attack can lead to state inconsistency and break confidentiality based on workflow 2 in Example 2.1.

State inconsistency: Transactions ④ and ⑤ represent the state change of U_A's balance where both \$100 and \$80, which have been signed by retailer DApp executors, are valid. After internally processing ⑤ and U_A receiving the product, a colluded executor tries to finalize U_A's balance still as \$100. However, L1 validators cannot detect the obsolescence, since they did not join in the workflow directly, leading to the state inconsistency between L1 and L2.

Break confidentiality: Suppose DApp executes transactions in a "black box", taking in encrypted U_A's balance and product price, then outputting a deducted balance (if sufficient). If one can rollback the system state and feed the "black box" with different transactions, the encrypted user balance can be detected by continually inputting dummy product price until the box outputs "insufficient".

To fulfill these requirements simultaneously, we introduce L2chain.

2.2 Layer-2 Blockchain Scaling

The L2 solution is built upon an L1 chain to reduce disseminating and consensus costs by off-loading transaction execution and part of order workloads to L2 nodes and only using the L1 chain for disputes [30]. Particularly, an accumulator (*e.g.*, Merkle tree) maintains system states and its digest of current state values (*e.g.*, Merkle tree root) denoted by D is recorded on the L1 chain. Users use D to authenticate if a state value is currently valid by checking its membership of D. To process transactions, there are three steps:

(1) **Off-chain execution.** L2 executors obtain the latest digest D and an ordered transaction batch B to perform a state transition $D' \leftarrow STF(D, B)$, producing an updated digest D'.
(2) **Execution correctness proof.** L2 executors then generate a proof $\pi \leftarrow PRF(D, B, D')$, proving that by executing transactions B, the state digest can be correctly updated to D'.
(3) **Finalization on-chain.** Finally, L2 executors invoke an L1 transaction, updating the on-chain state digest from D to D' with the proof π. According to the verification result through a function $\{0, 1\} \leftarrow VRF(D, B, D', \pi)$, L1 validators either commit the L1 transaction on-chain or abort it.

By doing so, user transactions only need to be transmitted and ordered among involved L2 executors and only one L1 transaction, representing an L2 batch execution result, is processed on-chain. It enhances the system by easing the consensus bottleneck.

2.3 Trusted Execution Environment (TEE)

Trusted Execution Environment (TEE) is a hardware solution providing an isolated memory area with the guarantee of *confidentiality* and *integrity* of data and codes running inside the area even if the entire platform is compromised. Examples are Intel's Software Guard Extensions (SGX) [5, 36, 46] and ARM TrustZone (TZ) [53].

Take SGX as an example. Every time to initialize the code runtime, SGX checks whether the hash of loaded code and data matches the developer's signature. During the run-time, SGX can prove to others that the specific code is executed correctly (*a.k.a. remote attestation* [5]). Specifically, a private key sk is embedded in each SGX enclave, and the corresponding public key pk is publicly known. Every time to produce an execution result of the loaded code, it signs with sk. Thus, remote nodes can use pk to verify the computation integrity. In addition, with the secretly embedded keys, TEE can also achieve *data sealing* which enables both the input and output data of the TEE can remain encrypted and only be decrypted inside a TEE ensuring the confidentiality of data as well.

2.4 RSA Accumulator

The RSA accumulator [17, 40] represents the members in a multiset S with a digest D where $D = g^{\Pi_{\forall s \in S} H_p(s)} \in \mathbb{G}$ and $H_p()$ is a hash-to-prime function. In particular, D is a member of an RSA quotient group [52] $\mathbb{G}$ ($\mathbb{Z}_N^{\times}/\{\pm 1\}$ where N is the production of two prime numbers) which has a fixed member g as the generator. Such that, the accumulator digest D is an integer with an unknown order in $\mathbb{G}$. The basic functions of an RSA accumulator include addition, removal, membership proof/verification and non-membership proof/verification of an element. Recently, Boneh *et al.* [11] expends the basic functions to support batch-based processing where the *Shamir's Trick* [58] plays an important role.

Shamir's Trick. Given a group element D and its x-th and y-th roots D^x, D^y, Shamir's Trick can be used to compute the (xy)-th root of D. We define $ST(D^x, D^y) \rightarrow D^{xy}$ where $ST()$ first computes the Bezout coefficients a and b of x and y, such that $ax + by = 1$, then outputs $(D^x)^b (D^y)^a$ as the result of D^{xy}. $ST(D^x, D^y)$ is more efficient than directly computing D^{xy} because it has fewer exponentiation operations.

Given the multiset S with digest D_t at time t, we now define the function of the RSA accumulator we need in this work as follows:

(1) $\boldsymbol{Add(D_t, S, E) \rightarrow \{D_{t+1}, S \bigcup E\}}$: add a batch E of elements into S and obtain the updated digest $D_{t+1} = D_t^{\Pi_{\forall e \in E} H_p(e)}$.
(2) $\boldsymbol{Del(D_t, S, E) \rightarrow \{D_{t+1}, S - E\}}$: delete a batch E of elements from S and obtain the updated digest $D_{t+1} = D_t^{1/\Pi_{\forall e \in E} H_p(e)}$.

(3) $\boldsymbol{MemP(D_t, S, E) \rightarrow w_E}$: a prover tries to convince to a verifier that all elements in E are valid members of S by providing a witness $w_E = g^{\Pi_{\forall e \in S, e \notin E} H_p(e)}$. Note that, w_E is exactly the same as D_{t+1} after deleting E from S.
(4) $\boldsymbol{MemV(D_t, w_E, E) \rightarrow \{0, 1\}}$: a verifier verifies if $E \subset S$ by checking if $(w_E)^{\Pi_{\forall e \in E} H_p(e)} = D_t$. Note that, with larger $\Pi_{\forall e \in E} H_p(e)$, *MemV* needs more exponentiation operations. However, the cost can be bounded within a constant by using the Wesolowski proof [60], for more details please refer to [11].

Keep witnesses. Both deletion and membership proof of an element e require to compute $H_p(e)$-th root of the digest D_t. However, without the RSA trapdoor, one has to reconstruct the digest from scratch *i.e.*, $D_{t+1} = g^{\Pi_{\forall s \in (S - \{e\})} H_p(s)}$ which is time-costly [11]. Thus, by pre-computing and storing the witness w_i ($\forall e_i \in S, w_i = g^{\Pi_{\forall e \in S, e \neq e_i} H_p(e)} = D_t^{1/H_p(e_i)}$), deletion and membership proof can be efficiently performed. Moreover, when elements of S change, the witness w_i of element e_i can be updated in the following cases:

(1) **A new element $e*$ is added to S:** The new witness is $w_i^{H_p(e*)}$.
(2) **An element $e*$ is deleted from S:** After the deletion of $e*$, we can obtain the updated digest $D_{t+1} = D_t^{1/H_p(e*)}$ of S. Thus, we can update each e_i's witness by $ST(w_i, D_{t+1})$.

2.5 Related Works

Next, we briefly review typical related DApp systems, from the perspectives of performance, confidentiality and security.

Performance. Existing DApp systems can be categorized in:

Permissionless: Ethereum [61], as the pioneer to support DApps with smart contracts, uses PoW [49] as the consensus protocol with only tens of tps. Improvements are achieved by layer-2 paradigm. Plasma [54] and zkSync [45] achieve hundreds of tps. However, their transaction validation strategies (*e.g.*, optimistic rollup [16] for Plasma and zero-knowledge proof [28] for zkSync) introduce high latency from tens minutes to even days.

Permissioned: Fabric [6] uses PBFT [18] and Quorum [19] uses IBFT [1] and Raft [51] to reach consensus among authorized nodes. Besides, protocols such as Tendermint [15], HotStaff [64], SBFT [31] also use message exchange to tolerant Byzantine faults (BFT) with hundreds of tps. However, existing systems aim to reach global consensus at once and do not provide flexibility for DApps to adopt self-desired protocols (*e.g.*, Paxos [38] and Raft for crash failure tolerance (CFT)), limit the overall performance.

Hybrid consensus: For flexibility, some systems allow multiple consensus protocols to coexist. Corda [56] supports pluggable protocols of DApps. While, CAPER [4], Multichain [29], cross-chain swap [34] and deal [35] maintain different views of chains with different protocols. However, the challenge is to deal with the cross-chain/app transactions which will be discussed later.

Confidentiality. Transactions on Ethereum are publicly available without confidentiality. Meanwhile, Plasma, zkRollup, and zkSync shift workloads to L2 centralized but untrusted processors. Although user transactions are private to L1 nodes, they are still available to L2 processors. Fabric introduces channels with Private Data Collections [7] to isolate DApps and their transactions. However, the channel structure is static, facing the challenge of cross-app transactions. While, Quorum, Corda, CAPER, Multichain, cross-chain swap, and deal make private transactions only visible to chosen participants. Precisely, transactions are encrypted or recorded in a private chain, and can only be decrypted or viewed by nodes with legitimate. However, in our motivation scenario, transaction executors, which are participants of the above systems, can also be malicious. Thus, our goal is more strict: making the private transactions only visible to authorized codes (SLAs).

Security. Typical systems such as Ethereum, Fabric, Quorum, Multichain, Corda and CAPER rely on consensus nodes to replay and validate transaction execution and order correctness. Meanwhile, L2 solutions embed the transaction batch execution and order results in one L1 transaction. It inevitably brings long latency to either run cryptography algorithms or wait for the protocol completion. Moreover, for cross-app transactions, tedious efforts are required. Systems that statically divide DApp states (*e.g.*, Fabric, Plasma, and zkSync) need to reform a new consortium or transform states between DApps. While, CAPER, cross-chain swap and deal use specially designed protocols to ensure the atomic transaction commit of cross-app transactions within a bounded time.

Our work is also related to TEE-based blockchains. For instance, Teechain [42] uses TEE to secure the payment network. [21] uses TEE to improve the efficiency of BFT consensus protocol. SlimChain [62] designs a stateless chain by using the TEE to decouple transaction execution and ordering, reducing the computation and storage bottleneck of the L1 validators. Although a stateless chain also executes transactions off-chain, each execution result still needs to be serialized by every validator which does not relieve the consensus bottleneck. Differently, an L2 architecture scales the system by improving the block space utilization. In addition, none of them focus on our motivation scenario, providing three features simultaneously. While [14] relieve the TEE rollback attack in BFT consensus-based systems only, which does not fit in our scenario where DApps can choose arbitrary consensus mechanisms.

In summary, in existing systems, DApps cannot share states flexibly with high throughput. Meanwhile, it lacks user privacy protection from malicious L2 nodes. Also, TEE-based chains cannot withstand the rollback attack well to retain transaction execution and order security. L2chain aims to address all of these issues.

3 L2CHAIN OVERVIEW

In this section, we first introduce our design goals, challenges and threat model. Then we provide an overview of the L2chain.

3.1 Design Goals and Challenges

We summarize our design goals with their technique challenges:

- **To achieve high performance (3 goals):**
  (1) Batch-based Layer-2 Processing: Transactions are processed in batches in an L2 network with execution results on-chain.
  (2) Balanced Storage: L1 validators only maintain cryptography digests of states to validate transactions execution results. Meanwhile, L2 executors only maintain states that they need.
  (3) Parallel Transaction Processing: DApp transactions without dependency on each other can be processed in parallel.
- **To achieve high confidentiality (1 goal):**
  (4) Confidential Transaction Execution: DApp transactions and private data are executed and maintained off-chain by related

L2 executors where L1 chain only records the state digest for integrity check. Meanwhile, DApp users can invoke confidential transactions, which even prevents DApp executors from detecting its specific operations on states.

- **To achieve high security (1 goal):**
(5) Secure Off-chain Execution: Every L2 executors can correctly execute a transaction batch in a predetermined order to guarantee system security and withstand the rollback attack.

Threat Model: Note that nodes in both layers of L2chain may perform maliciously. Here, we introduce our threat model.

Malicious Layer-1 Validators: As the maintainer of the ledger recording system state digest, a validator may be crash or malicious to subvert the ledger consistency. Thus, L2chain inherits existing blockchain design with consensus protocols to withstand malicious validators. Moreover, as validators only process state digests, no private information is leaked to validators.

Malicious Layer-2 Executors: There are two malicious behaviors: 1) deviating from the SLAs (executing wrong codes) to propose incorrect results poisoning the on-chain digest. 2) do not follow the correct order to execute transactions (*i.e.*, rollback attack), causing L1 and L2 state inconsistency and breaking the user privacy (as illustrated in Sec. 2.1). Existing designs cannot prevent such malicious L2 executors, which is our primary effort.

Challenges: To achieve our design goals, we face three challenges. Firstly, L1 validators no longer have system states and actual L2 transaction contents. Thus, when DApps parallelly propose execution results of transaction batches, it is challenging for validators to resolve the update contention on the on-chain state digest where potential conflict may exist. Secondly, to keep transactions private from both L1 and L2 nodes, as shown in Sec. 2.1, it is challenging to overcome the rollback attack, breaking the confidentiality guarantee even with an execution black box (*e.g.*, TEE). The third challenge is ensuring state consistency between L1 and L2 networks. Because transactions are only executed by involved DApps in an L2 network, it is challenging to synchronize the transaction execution order in both L1 and L2 networks to produce consistent states.

3.2 Current Limitations

In this section, we analyze the limitations of the current L2 architecture and propose our solution roadmap.

Current L2-based solution would be: 1) Executors obtain the latest L1 states and an L2 transaction batch; 2) Order and execute the batch off-chain with encrypted inputs in a TEE to update states with an execution proof. 3) Writes an L1 transaction, stipulating that the result can only be committed if no intermediate update to involved states. However, two aspects limit this solution:

Aspect 1: Space Cost and Data Contention. Most L2 systems accumulate states into a Merkle Patricia Trie (MPT) [61] shown in Fig. 2 where an extension or branch node records an address prefix, indexing to states, and a leaf node stores the state value. Meanwhile, each node contains a hash value of its serialized child node hashes. Each L1 block has an MPT root, acting as the state digest. L1 validators maintain the digest in two ways:

Stateful validators (with high space cost) to store all states (encrypted if privacy is required) and construct the latest world state digest for each block. Thus, in step 3, L2 executors must include previous and updated values of each affected state in an L1 transaction for validators to further process, exposing high space costs. With a limited block capacity, the throughput improvement is marginal compared with a pure L1 chain. Because due to the consensus bottleneck restricting committed blocks per second, the more committed transactions in a block, the higher the throughput is.

Stateless validators (exposing high data contention) to stores the latest digest only [11, 12, 28, 62]. Executors process L2 batches and include the updated digest in a succinct L1 transaction, dramatically improving block space utilization. However, widely used MPT accumulators can cause high data contention on the digest (MPT root) and force processing L2 batches serially. For example, in Fig. 2, states with values $v_1 \backsim v_3$ belong to DApps $A_1 \backsim A_3$ respectively. When updating v_1, to obtain updated MPT root, A_1 executors require n_6's latest hash. Thus, A_1 and A_2 cannot independently access n_5 and n_6 in parallel which will produce two conflicting roots.

For high performance, an ideal way is to remain validators stateless and reduce the state digest contention. Our idea is to organize states in an RSA accumulator and let executors proactively split the digest of affected states in an L2 batch from the world state digest, then merge the updated digest back after processing. We call it split-execute-merge (SEM) and detail in Sec. 4 with the optimization to minimize the RSA witness generation cost for executors in Sec. 5.

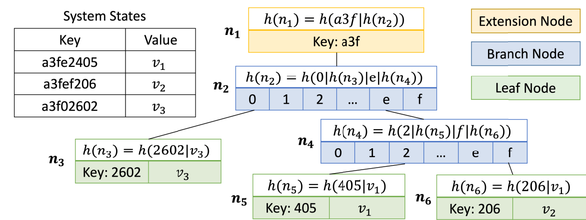


Figure 2: An Example of the Merkle Patricia Trie

Aspect 2: Uncontrollable Execution Order. The more critical problem is that the executor can execute user transactions without any audit or restriction. Existing L2 solutions follow the Order-Execute-Validate (OEV) processing model to order transactions on-chain first and let replicas execute and validate results to update local states. However, there is no strict restriction to prevent executors from deviating from the protocol in the execution phase. As described in Sec. 2.1, a malicious executor can replay the execution based on the same input states with uncommitted artifact transactions in an arbitrary order to detect the TEE side information. Such a rollback attack can easily break transaction privacy.

To overcome this issue, based on the SEM architecture, we propose *two-step execution* in Sec. 6 to first determine input states and finalize the transaction order on the L1 chain, then restrict an executor can only execute ordered transactions in a TEE.

3.3 System Overview

Fig. 3 shows the basic architecture of the L2chain.

Layer-1 blockchain. As the fundamental part of L2chain, an L1 blockchain maintains the authentication of system states shared by DApps built upon the chain. In addition, *L1 validators* maintain the chain ledger by appending blocks with *L1 transactions* through consensus protocols. We adopt the account-based model [61] where

each state is an address-value pair. In the rest of paper, we use S_i to denote a state address and $S_i.v$ to be its value. We remain L1 chain to be stateless and organize states into an RSA accumulator introduced in Sec. 2.4 with its digest recorded on-chain. Compared with the MPT, the RSA accumulator eliminates the unnecessary structural information of the data and can be easily divided into multiple homogeneous accumulators of state subsets. Specifically, given a state set S, for all $S_i \in S$, we use a hash-to-prime function $H_p()$ to convert the address-value pair $\{S_i, S_i.v\}$ into a prime number and add it into the RSA accumulator with digest $D = g^{\prod_{\forall S_i \in S} H_p(\{S_i, S_i.v\})}$.

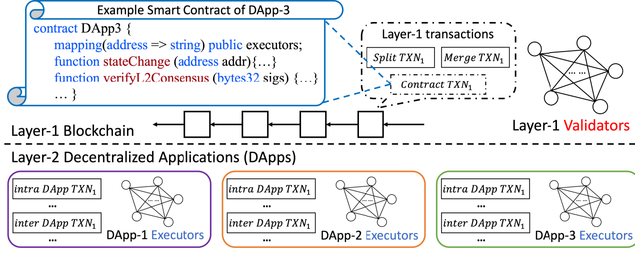


Figure 3: L2chain System Architecture

Layer-2 decentralized applications. The L2 network consists of multiple DApps established by deploying smart contracts with service-level agreements (SLAs) on-chain by L1 transactions (we do not specially discuss the DApp initialization transaction in the rest of the paper for concise). In particular, SLAs stipulate the business logic and which on-chain states can be accessed or updated by a DApp. Besides, the contract also records what consensus protocols to use and how to verify the consensus result[1], represented by executable codes and exposing as a contract ABI, for specified intra- and inter- DApp operations. Meanwhile, each L2 DApp executor equips with TEE-enabled CPUs and maintains (encrypted in privacy protection mode) state values that the executor can access.

Layer-1 and Layer-2 Transactions. We divide transactions into L1 and L2 based on where they are processed. We first define the L2 DApp transaction generated by DApp users, whose processing throughput and latency are the main criteria of the system.

Definition 3.1. ***Layer-2 DApp transaction*** *is denoted by a tuple* $tx = (addrs, op, \sigma)$ *where* $addrs$ *represents input state addresses.* op *is the operation on the inputs and* σ *is the initiator signature.*

A DApp transaction can be *intra-* or *inter-* DApp, depending on whether the read/write sets and involved executors are within one DApp or across multiple DApps. Compared with intra-DApp transactions which can be directly processed within a DApp, processing inter-DApp transactions has two distinct steps. ❶ ***Deploy cross-app SLAs on the L1 chain.*** To enable a new paradigm of inter-DApp transactions, new SLAs must be registered on the L1 chain where involved DApps jointly establish a smart contract, stipulating which states can be accessed by each DApp and what is the L2 network consensus protocol, etc.. ❷ ***Execute inter-DApp transactions in the L2 network.*** We detail the differences between the intra- and inter-DApp transaction execution in Sec. 6.

The execution results and corresponding correctness proofs of off-chain processed L2 transaction are collected in L1 transactions for L1 validators to update the blockchain ledger. Specifically, L1 transactions are categorized into ***split*** and ***merge*** transactions to support our SEM workflow which will be detailed in Sec. 4.

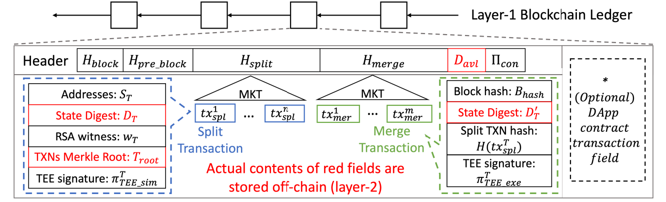


Figure 4: Layer-1 Blockchain Structure

Layer-1 Blockchain Structure. Fig. 4 shows the L1 ledger structure where L1 transactions are organized into linked blocks with a fixed capacity. Specifically, each block contains the content hash of itself and its previous block. Π_{con} indicates the block has be consented by L1 validators. For instance, if PoW is the L1 consensus protocol, Π_{con} is the block nonce and for PBFT/Raft, Π_{con} is the aggregated signatures of validators. For L1 split and merge transactions, we organize them into two Merkle trees and record the Merkle hash root H_{split} and H_{merge} in the block header. In addition, each block contains an RSA accumulator digest D_{avl}, representing the system states S_{avl} that are currently NOT accessed by any DApp (detailed in Sec. 4). Besides, a block also has an optional field for DApp smart contracts and corresponding deploying transactions.

4 SPLIT-EXECUTE-MERGE WORKFLOW

In this section, we introduce the split-execute-merge (SEM) workflow to process transactions in L2chain based on the Fig. 5.

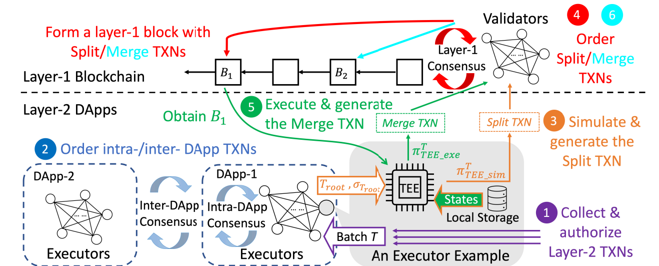


Figure 5: L2chain transaction processing workflow

(1) A DApp user invokes an L2 transaction tx which is authenticated by DApp executors and collected into a batch T.
(2) DApp executors order transaction batch T within involved DApp nodes through the predetermined consensus protocol. Then organize T in a Merkle tree with root T_{root} and obtain an aggregated signature $\sigma_{T_{root}}$ on T_{root} signed by executors.
(3) DApp executors simulate T in the TEE to obtain its read/write sets S_T with a TEE signature $\pi^T_{TEE_sim}$, then generate an L1 split transaction tx^T_{spl} to split the digest D_T of S_T from the digest D_{avl} (in the latest L1 block) of available L1 states S_{avl}.
(4) After tx^T_{spl} appearing in an L1 block B, L2 executors can obtain B's block header H_B with L1 consensus proof Π_{con}.
(5) With the input of T, T's read/write set S_T and H_B, L2 executors execute T and obtain the updated digest D'_T of S_T with the TEE execution correctness proof $\pi^T_{TEE_exe}$.

[1] E.g., for communication-based protocols, it is to verify the participants' signatures.

(6) Executors generate an L1 merge transaction tx^T_{mer}, containing the hash of block B with tx^T_{spl}, updated digest D'_T and TEE signature $\pi^T_{TEE_exe}$. After validation, L1 validators record tx^T_{mer} on-chain which completes the process.

In particular, the entire workflow is divided into three phases.

Split-phase (steps 1∽3): L2 executors collect L2 transactions and categorize them into intra- and inter- DApp. In each period, executors order different transactions separately among involved DApp executors by running consensus protocols determined in the on-chain SLAs to form an L2 transaction batch. Ordered batch is organized into a Merkle tree to preserve the order information, and involved nodes sign on the tree root as the consensus evidence. Then, executors simulate the transaction batch in the TEE to obtain read/write sets with the TEE signature $\pi^T_{TEE_sim}$ (detailed in Sec. 6), based on which generate an L1 split transaction defined as follows:

*Definition 4.1 (**Split transaction**).* Given an ordered layer-2 transaction batch T organized in a Merkle tree with root T_{root}, its read/write set S_T signed by TEE with $\pi^T_{TEE_sim}$, and the current available layer-1 states S_{avl} with digest D_{avl}, a split transaction is defined as a tuple $tx^T_{spl} = (S_T, D_T, w_T, T_{root}, \pi^T_{TEE_sim})$ where $D_T = \prod_{\forall S_i \in S_T} H_p(\{S_i, S_i.v\})$, $w_T = MemP(D_{avl}, S_{avl}, S_T)$.

The split transaction tx^T_{spl} is then sent to L1 validators who check: 1) if the DApp has the authority to access states in S_T according to contract SLAs. 2) if $D_{avl} = exp(w_T, D_T)$, which is the membership check that S_T is in the current available state values S_{avl} (*i.e.*, $MemV(D_{avl}, w_T, S_T)$). If all checks pass, validators packs tx^T_{spl} in an L1 block. As discussed in Sec. 2.4, generating the witness w_T for S_T can be computation and storage costly. Thus, in Sec. 5, we propose *witnesses cache* to minimize such an overhead cost.

Conflict elimination: As DApps concurrently process L2 transactions, conflicts may occur in two aspects: ❶ *Across L2 batches* where two L1 split transactions of two L2 batches try to split the digests consisting of the same state simultaneously. Such conflicts are resolved by L1 validators. Specifically, in each L1 block, each available L1 state can only be split once (appear in one split transaction). Otherwise the entire block is invalid, making it nonprofitable for the validator. Meanwhile, in systems with non-deterministic L1 consensus protocols, the issue that two validators generate conflicting blocks based on the same available L1 states is similar to the public chain fork problem which can be resolved by the longest chain rule [49]. ❷ *Within an L2 batch* where two L2 transactions have conflicts (*e.g.*, an inter-DApp transaction reading the state updating by an intra-DApp transaction). Since each L2 batch is still serially processed in the L2 network, to order an L2 batch in the split phase, each transaction is simulated (detailed in Sec. 6) and consent among related executors to prevent invalid state access.

Execute-phase (steps 4 and 5): After the split transaction tx^T_{spl} of states S_T has been recorded in an on-chain block B, layer-2 executors can obtain the block header H_B. In particular, the valid tx^T_{spl} is essential to execute transactions, thus we need to check: 1) the validity of layer-1 consensus proof Π_{con}, indicating B is a valid on-chain block, where Π_{con} is determined by the layer-1 consensus protocol (detailed in Sec. 6). 2) If tx^T_{spl} is a valid member of B, by checking the Merkle hash root H_{split} of all split transactions in B.

We leverage the TEE to execute transactions. Specifically, the inputs of TEE are transactions T, the read/write sets S_T, the split transaction tx^T_{spl} of T, and the header H_B of block containing tx^T_{spl}. Then TEE executes transactions to update states in S_T to S'_T (detailed in Sec. 6). Finally, TEE outputs the digest $D'_T = \prod_{\forall S_i \in S'_T} H_p(\{S_i, S_i.v\})$ of updated states S'_T and a proof $\Pi^T_{TEE_exe}$, indicating transactions T are correctly executed to obtain D'_T.

Merge-phase (step 6): To complete the SEM process, executors generate an L1 merge transaction to merge the digests of updated states in S'_T and current L1 available states to make S'_T be accessed by other DApps. We define the merge transaction as follow:

*Definition 4.2 (**Merge transaction**).* Given the hash B_{hash} of block containing the split transaction tx^T_{spl} of ordered transactions T, and the updated digest D'_T with its execution correctness proof $\pi^T_{TEE_exe}$, a merge transaction is a tuple $tx^T_{mer} = \{B_{hash}, H(tx^T_{spl}), D'_T, \pi^T_{TEE_exe}\}$ where $H(tx^T_{spl})$ is the hash of tx^T_{spl}.

Layer-1 validators process the merge transaction tx^T_{mer} in following steps: 1) obtain tx^T_{spl} from the block with hash B_{hash} and check its integrity through $H(tx^T_{spl})$. 2) verify $\pi^T_{TEE_exe}$ based on $H(tx^T_{spl})$ and D'_T. 3) if all checks pass, pack tx^T_{mer} into a layer-1 block with the updated available states digest D_{avl} (detailed next).

Aggregate multiple split and merge transactions: A layer-1 block can contain multiple split and merge transactions. Thus, their effects to the current available states digest D_{avl} need to be aggregated and recorded in the block. Specifically, for two split transactions tx^1_{spl} and tx^2_{spl}, a validator checks if $tx^1_{spl}.S_1 \bigcap tx^2_{spl}.S_2 = \emptyset$. If not, it means state contention exists. Thus, one of them is discarded randomly[2]. After resolving all state contentions of split transactions, we continue updating D_{avl} by $D_{avl} := ST(D_{avl}, tx^i_{spl}.w_T)$ for each split transactions tx^i_{spl}, and $D_{avl} := exp(D_{avl}, tx^j_{mer}.D'_T)$ for each merge transaction tx^j_{mer}. Finally, the validator organizes split and merge transactions into two Merkle trees with hash roots H_{split} and H_{merge} separately, and record D_{avl} in the block as well.

Pipeline to reduce the latency: Compared with the pure L1 architecture, SEM introduces an additional on-chain transaction to commit the L2 batch execution results, which brings extra latency, exacerbating data contention and delaying successive batch processing. To relieve this issue, we can pipeline the digest split and merge phases. Specifically, by allowing transmitting merge transactions in the L2 network, executors can combine the merge of updated digests with the split of to-be-processed state digest into one on-chain transaction. We detail in an example as follows:

Example 4.3. Suppose current L1 available digest D_{avl} represents states S_1∽S_8, then executors in $DApp_1$ split S_1∽S_3 with digest D_1 (now L1 available digest becomes D^1_{avl} where $D_{avl} = exp(D^1_{avl}, D_1)$) and process an L2 batch T_1 to update S_1∽S_3 with the outcome digest D^*_1. With pipeline, instead of $DApp_1$ executors invoke an L1 merge transaction $tx^{T_1}_{mer}$ for D^*_1 immediately, they propagate $tx^{T_1}_{mer}$ in the L2 network. Assume now an L2 batch T_2 of $DApp_2$

[2]For simplicity, in this work, we do not consider concurrent state access.

needs to access S_1 and S_8. Once receiving $tx_{mer}^{T_1}$, $DApp_2$ executors can recover the latest L1 available digest $D_{avl}^2 = exp(D_{avl}^1, D_1^*)$ and generate a split transaction $tx_{spl}^{T_2}$ for S_1 and S_8. Finally, $DApp_2$ executors invoke one L1 transaction, combining $tx_{mer}^{T_1}$ and $tx_{spl}^{T_2}$.

5 WITNESSES CACHE AND OPTIMIZATION

As we discussed in Sec. 4, given the current available states S_{avl} with the digest D_{avl}, to process a transaction batch T ,accessing states S_T, layer-2 executors needs to provide the witness $w_T = MemP(D_{avl}, S_{avl}, S_T)$ which is the membership proof of $S_T \subset S_{avl}$. However, existing witness generation methods cannot achieve high efficiency in both computation and storage.

Specifically, one way to generate w_T is to reconstruct the digest of states in $S_{avl} - S_T$ (*i.e.*, $g^{\Pi_{\forall S_i \in (S_{avl} - S_T)} H_p(\{S_i, S_i.v\})}$). This method not only breaks the privacy where executors need to keep all address-value pairs, including states without the access permissions, but also has high computation cost where exponentiation operations are proportional to the number of system states.

The other way is to pre-compute and cache the membership witness $w_i = D_{avl}^{1/H_p(\{S_i, S_i.v\})}$ based on the latest D_{avl} for each state S_i and the w_T is computed by recursively performing the Shamir's Trick for all w_i where $S_i \in S_T$ (*i.e.*, $\forall S_i \in S_T, w_T = ST(w_T, w_i)$). However, caching witness for every state is storage-costly. What's worse, every time D_{avl} changes due to a newly committed L1 block, all witnesses need to update which is computationally heavy. To reduce the overhead cost, we introduce the witnesses cache mechanism for L2 executors in L2chain.

5.1 Witness Cache

Our basic idea is to let DApp executors to individually partition their accessible system states into a constant number of groups based on their access frequency f_i (*e.g.*, times per block) to each state S_i. For each group of states, executor maintains a witness, which is an integer in the RSA group. For a group of states S' with witness $w_{S'}$, the witness w_i of each individual state $S_i \in S'$ can be obtained by $w_i = w_{S'}^{\Pi_{\forall S_j \in S', S_j \neq S_i} H_p(S_j)}$. By doing so, the cached witnesses can have bounded storage and update costs. We then formally define the witness cache as follows:

*Definition 5.1 (**Witness Cache**).* We partition system states into τ groups denoted by C_i ($i \in [1, \tau]$) where τ is an executor-defined constant and the index i denotes the ascending order of the group cardinality. Given the digest D_{avl} of the current available states, the executor caches a witness $w_{C_i} = D_{avl}^{1/\Pi_{\forall S_j \in C_i} H_p(S_j)}$ for each group.

Update Witnesses. To update cached witnesses after D_{avl} changes due to a newly appended L1 block, we first deal with L1 split transactions, splitting S_T with digest D_T and witness w_T from S_{avl}:

(1) For a cache group C_i where $C_i \bigcap S_T = \emptyset$, the updated witness is computed as $w_{C_i} = ST(w_{C_i}, w_T)$.
(2) For a cache group C_i where $C_i \bigcap S_T = S' \neq \emptyset$, the updated witness is computed as $w_{C_i} = ST((w_{C_i})^{\Pi_{\forall S_i \in S'} H_p(\{S_i, S_i.v\})}, w_T)$.

Then we deal with the merging of S_T with digest D_T to S_{avl}.

(1) For a cache group C_i where $C_i \bigcap S_T = \emptyset$, the updated witness is computed as $w_{C_i} = exp(w_{C_i}, D_T)$.
(2) For a cache group C_i where $C_i \bigcap S_T = S' \neq \emptyset$, the updated witness is computed as $w_{C_i} = exp(w_{C_i}, D_T / \Pi_{\forall S_i \in S'} H_p(\{S_i, S_i.v\}))$.

Note that, the **lazy loading strategy** can also prevent frequent cache updates where executors aggregate updated digests and witnesses in each block and refresh a witness cache only when accessed. Here we provide a concrete example of the witness cache:

Table 1: Example of states with access frequencies (/block).

S_1:0.3	S_2:0.2	S_3:0.1	S_4:0.5	S_5:0.3	S_6:0.9	S_7:0	S_8:0.5
S_9:0.8	S_{10}:0.3	S_{11}:0	S_{12}:0.1	S_{13}:0	S_{14}:0	S_{15}:0	S_{16}:0.6

Cache	C_1			C_2				C_3				C_4				
Witness	$D_{avl}^{1/\prod_{i=1}^{3} H_p(\{S_i, S_i.v\})}$			$D_{avl}^{1/\prod_{i=4}^{7} H_p(\{S_i, S_i.v\})}$				$D_{avl}^{1/\prod_{i=8}^{11} H_p(\{S_i, S_i.v\})}$				$D_{avl}^{1/\prod_{i=12}^{16} H_p(\{S_i, S_i.v\})}$				
State Content	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}

Figure 6: A feasible witness cache organization

Example 5.2. Table 1 shows the access frequencies to states S_1 to S_{16} of a DApp executor who sets $\tau = 4$. A feasible cache organization is shown in Fig. 6. Suppose currently all states are available and $D_{avl} = g^{\Pi_{i=1}^{16} H_p(\{S_i, S_i.v\})}$. Assume a new layer-1 block, containing a split transaction tx_{spl}^T where $tx_{spl}^T.D_T = H_p(\{S_1, S_1.v\})$ and $tx_{spl}^T.w_T = D_{avl}^{1/H_p(\{S_1, S_1.v\})}$, splits S_1 from the available states. The executor can update the cache by: 1) for C_1, $w_{C_1} = (w_{C_1})^{H_p(\{S_1, S_1.v\})}$. 2) for C_i in $C_2 \sim C_4$, $w_{C_i} = ST(w_{C_i}, D_{avl}^{1/H_p(\{S_1, S_1.v\})})$. Later, a merge transaction tx_{mer}^T with the updated digest D_T' of S_1 is on-chain. Executor updates the cache for all C_i by: $w_{C_i} = (w_{C_i})^{tx_{mer}^T.D_T'}$.

5.2 Witness Cache Optimization

We now consider the optimization of witness cache. Specifically, while the witness storage and update costs are bounded by the cache group count τ, the cache organization can affect the average witness generation cost. Because, each executor has different access frequencies to each state. Thus, in this subsection, we define the witness cache optimization problem for each individual executor to organize their local cache to minimize the average witness generation cost when generating split transactions.

Average witness generation cost is measured by the summation of the executor access frequency to each state times the number of required exponentiation operations to generate the membership witness for each state (*i.e.*, cache group cardinality minus one).

For instance, in Example 5.2, to generate the membership witness for S_1, the executor needs to compute $(w_{C_1})^{H_p(\{S_2, S_2.v\}) H_p(\{S_3, S_3.v\})}$ with two exponentiation operations, determined by $|C_1| - 1$. Thus, the average witness generation cost of the executor is computed as $2 * (0.3 + 0.2 + 0.1) + 3 * (0.5 + 0.3 + 0.9 + 0) + 3 * (0.5 + 0.8 + 0.3 + 0) + 4 * (0.1 + 0 + 0 + 0 + 0.6) = 13.9$. Our goal is to minimize such a cost. Thus, we define the witness cache optimization problem as:

*Definition 5.3 (**Witness cache optimization problem.**).* Given the executor's access frequency f_i to each state S_i, the number of witness cache groups τ, our goal is to determine a partition of states into τ caches such that to minimize $\sum_{i=1}^{\tau} (|C_i| - 1) \cdot \sum_{\forall S_j \in C_i} f_j$.

Optimal cache group count τ. Before giving the solution to the optimization problem, we first discuss how to determine the

optimal cache group count τ. Suppose there are n accessible states for an executor and the cost of adding one state into the digest is α. Without the witness cache, the average time cost to generate the RSA membership witness for states is $cost_1 = \alpha(n-1)\sum f_i$. On the other hand, with τ witness caches, the amortized witness generation cost can be bounded by $\alpha(\frac{n}{\tau}-1)\sum f_i$. Meanwhile, suppose the cost of updating one witness based on one on-chain block is β, the average cache update cost is $\beta\tau$. In total, the average cost of using the witness cache is $cost_2 = \alpha(\frac{n}{\tau}-1)\sum f_i + \beta\tau$. Thus, to find the optimial cache group count is to maximize $cost_1 - cost_2 = \alpha(n-1)\sum f_i - \alpha(\frac{n}{\tau}-1)\sum f_i - \beta\tau$ which is achieved when $\tau = \sqrt{\frac{\alpha n \sum f_i}{\beta}}$.

Dynamic programming-based algorithm. To solve the witness cache optimization problem, the basic idea is to sort states based on the access frequency in descending order and use dynamic programming to determine the optimal partition points to form τ cache groups. We first propose a lemma.

Lemma 5.4. *For the optimal partition of the ordered witness caches $C_i (i \in [1,\tau])$, $\forall |C_i| < |C_j|$, the smallest state access frequency in C_i is greater than or equal to the highest frequency in C_j.*

Due to limited space, we refer the proof to our technical report [63]. With this lemma, we propose a dynamic programming-based algorithm to optimize the cache organization:

Algorithm details: As shown in Algo. 1, we sort states based on access frequencies. Then we create a 3D (i, j, k) array with ∞ as the default value (line 1). These dimensions represent: i: the state list position in making partition decision, j: cardinality of the largest cache group, k: the number of unformed caches. Values of the dimension $k = 0$ are initialized in line 3, calculated by the witness generation cost of the last cache group. Then, we recursively compute all entries starting from the dimension $(0, j, k)$ and find the minimum value as the optimal objective value (lines 5-9). Each time, we compare the cost to decide whether to form a new cache group including states from i to $i + j$. Finally, $OutputPartition()$ can produce the optimal partition based on the computed table $t(i, j, k)$. Due to space limit, we omit the details.

Complexity analysis. To sort n states based on the access frequency, the time complexity is $O(n \log n)$. To compute all entries of the DP table, it takes $O(n^2\tau)$. Thus, the overall time complexity of Algo. 1 is $O(n^2\tau)$.

Dynamic change of access frequencies. Since the access frequency of executors to each state can change dramatically after a period, we need to re-organize the cache to minimize the average witness generation cost as well. To handle the dynamic issue, for each cache group C_i order by the cardinality in accending order, we record its cardinality $|C_i|$, highest $high(C_i)$ and lowest $low(C_i)$ access frequencies of its inside states as well as the average access frequency $\bar{f}$ of all states. When access frequencies of states S^* change, we first reassign S^* to caches such that $\forall S_i \in C_j, f_i \in [low(C_j), high(C_j)]$ holds. Then, starting from C_1 to C_τ, for each C_i, we continue reassigning the states in C_i with lowest access frequency to C_{i+1} until $\forall S_j \in C_i, (|C_i|-1)\sum f_j \le \frac{n}{\tau}(\frac{n}{\tau}-1)\bar{f}$. The time complexity of such method is $O(m\tau \log \frac{n}{\tau})$ where m is the number of updated access frequencies, which is far less than repartition all caches from scratch. Similar to the analysis in the optimal cache group count selection, with α as the cost of adding one element into the accumulator, such reassignment strategy ensures that the upper bound of the average witness generation cost is $\alpha(\frac{n}{\tau}-1)\sum f_i$ which is the exception cost by using the cache mechanism. Meanwhile, executors can also choose to rerun the Algo. 1 periodically to seek for the optimal cache organization.

Algorithm 1: DP-based cache organization optimization

Input : State addresses $S_i, i \in [1,n]$ with access frequency f_i, cache group count τ.

1. Sort S_i based on f_i in the descending order;
2. t(i, j, k) ← initialize a $n \times n \times \tau$ 3D array with ∞ by default;
3. **for** $i \in [1,n], j \in [1,n]$ **do**
4. t(i,j,0) = $(n-i-1)\sum_{\forall l \in [(n-i),n]} f_l$;
5. **for** $k \in [1,\tau]$ **do**
6. **for** i *from* $n-2$ *to* 0 **do**
7. **for** j *from* $\frac{n-i}{k+1}$ *to* 0 **do**
8. partition = t(i+j, j, k-1) + $(j-1) \cdot \sum_{l=i}^{i+j} f_l$;
9. non_partition = t(i, j+1, k);
10. t(i, j, k) = min(partition, non_partition);
11. opt-cost = find minimum value in $t(0,j,\tau), \forall j \in [1,n]$;
12. *OutputPartition()*;

6 LAYER-2 TWO-STEP EXECUTION

In this section, we detail the two-step L2 DApp transaction execution in the SEM process. It aims to prevent the rollback attack, resulting in transaction content leakage and state inconsistency.

Basic idea: As described in Sec. 3.2, we divide the execution into two steps. The first step is *simulation* in the split-phase. Since we target an account-based model where general-purpose transactions can affect arbitrary states and be encrypted for privacy, their read-/write sets cannot be known in advance. Meanwhile, we need to enforce TEEs only executing ordered transactions based on committed input states to prevent the rollback attack. Thus, we use the simulation step to obtain read/write sets and finalize the order of transaction batch in the L2 network to produce a corresponding L1 split transaction tx_{mer}^T. Once tx_{mer}^T is committed on-chain, the execute-phase is for executors to *execute* T to obtain the result.

Algorithm 2: Transaction simulation (in TEE)

Input : Ordered transactions T with its Merkle root T_{root} and aggregated signature σ_{Troot}, local states S.

1. **if** $verify(T_{root}, \sigma_{Troot})$ $failed$ **then abort**;
2. **if** $order_check(T, T_{root})$ $failed$ **then abort**;
3. $S_T \leftarrow simulate(T, S)$;
4. $D_T, \neg D_T \leftarrow witness_generation(S_T)$; ***// see Sec. 5***
5. $\pi_{TEE_sim}^T \leftarrow TEE.sign(< T_{root}, S_T, D_T, \neg D_T >)$;
6. **return** $< S_T, D_T, \pi_{TEE_sim}^T >$

Step 1: Simulation. As introduced in Sec. 4, during the split-phase, involved DApp executors order an L2 transaction batch T by running the predetermined L2 consensus protocol and organize T into a Merkle tree with root T_{root}. Algo. 2 shows the simulation step where TEE first verifies the input validity (lines 1-2) by checking:

1) if T is consented by related executors through the aggregated signature σ_{Troot}; 2) if input T is in a correct order by reconstructing the transaction Merkle tree and check if the produced root matches the signed root T_{root}. Then, TEE simulates T based on the local states to obtain the state addresses S_T to be read/wrote (line 3). Next, the executor generates the digest D_T of S_T and the witness $\neg D_T$, indicating S_T is a valid member of current available L1 states (line 4). Note that, line 4 can be done outside the TEE. Finally, TEE signs on the tuple of $< T_{root}, S_T, D_T, \neg D_T >$ as the authority proof.

Step 2: Execution After the split transaction tx^T_{spl} of T is committed on-chain in a block with header H_B, executors can execute T to update states. As shown in Algo. 3, executors first verifies the input validity (lines 1-4) by checking: 1) if the block B with content $B_{content}$ is valid through Π_{con}. For instance, H_B contains three hash values $H(B)$, $H(B')$ and $H(B_{content})$ where B' is B's previous block. In a PoW-based L1 chain, H_B also contains the PoW difficulty $diff$ and a nonce, regarded as the Π_{con}, provided by the validator. TEE verifies Π_{con} by checking if $H(B){\equiv}H(H(B')|H(B_{content})|\Pi_{con})) \leq diff$. Meanwhile, in a PBFT or Raft-based chain, Π_{con} is an aggregated signature signed on H_B by validators which can be directly verified in a TEE; 2) if the split transaction tx^T_{spl} is a valid member of the block through Merkle branch proof; 3) if T is in a correct order by the same way in the simulation step. If all checks pass, TEE executes T on states $S_T \in tx^T_{spl}$ and obtain the updated state digest D'_T (lines 5-6). Finally, TEE signs on the tuple of split transaction hash and updated digest, proving the execution correctness and the signature is used to generate a merge transaction.

Algorithm 3: Transaction execution (in TEE)

Input : Ordered transactions T with its split transaction tx^T_{spl}, and corresponding block header H_B.

1. $\Pi_{con}, H_{split} \leftarrow H_B$; **if** Π_{con} *is invalid* **then abort**;
2. **if** $integrity_check(tx^T_{spl}, H_{split})$ *failed* **then abort**;
3. $S_T, T_{root} \leftarrow tx^T_{spl}$;
4. **if** $order_check(T, T_{root})$ *failed* **then abort**;
5. $S'_T \leftarrow execute(T, S_T)$;
6. $D'_T \leftarrow g^{\prod_{\forall S_i \in S'_T} H_p(\{S_i, S_i.v\})}$;
7. $\pi^T_{TEE_exe} \leftarrow TEE.sign(< H(tx^T_{spl}), D'_T >)$;
8. **return** $< D'_T, \pi^T_{TEE_exe} >$

Inter-DApp transactions: As TEE provides a trusted subsystem, preventing replicas from equivocating, we adopt the hybrid fault model consensus [10, 20, 39] in the cross-DApp simulation and execution protocol to process the inter-DApp transaction tx in a batch T. Fig. 7 is an example with DApps $A_0 \backsim A_2$ where A_0 and A_2 have one executor $E_{0,0}$ (prenominated as the *coordinator*) and $E_{2,0}$ respectively, and A_1 has three executors $E_{1,0} \backsim E_{1,2}$. ❶ *Prepare:* The coordinator sends tx to all involved executors. ❷ *Commit:* Each $E_{i,j}$ simulates/executes tx based on local values of states $S_{i,j}$ in A_i and broadcasts the tuple $<S_{i,j}, D_{i,j}, \pi_{i,j}>$ where $D_{i,j}{=}\prod_{\forall s \in S_{i,j}} H_p(\{s, s.v\})$ and $s.v$ is the original/updated value of s in the simulation/execution step, and $\pi_{i,j}$ is the TEE signature. ❸ *Reply:* Each $E_{i,j}$ verifies if $\forall A_i$, $> \frac{1}{2}$ executors commit the same $S_{i,j}$ and $D_{i,j}$. If so, $E_{i,j}$ replies to the coordinator and if in the execution step, $E_{i,j}$ updates its local states. ❹ *Aggregate:* Once the coordinator receives replies from all DApps, to generate a split/merge transaction, it obtains $S_T{=}\bigcup S_{i,j}$, $D_T/D'_T = \prod D_{i,j}$ and $\pi^T_{TEE_sim}/\pi^T_{TEE_exe}$ by aggregating $\pi_{i,j}$ in simulation/execution steps (*e.g.*, Boneh-Lynn-Shacham (BLS) [44], an aggregation-friendly signature, can be used). Note that, to recover from a failure coordinator, a *view change protocol* is required for others to replace the failure coordinator. Due to the limited space, we refer more details in [10].

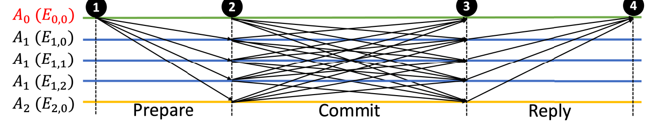


Figure 7: Cross-DApp simulation and execution protocol

Security Analysis. Next, we analysis how the two-step execution mechanism can prevent the rollback attack.

For the Deterministic Layer-1 Consensus (**e.g.*****, PBFT/Raft)*** where after reaching consensus, no fork from the newly appended block can be generated anymore, the rollback attack is strictly prevented. Suppose the batch T has been executed once based on read/write set S_T with digest D_T recorded in an on-chain split transaction tx^T_{spl}. To obtain a deviated execution result, attackers must generate another split transaction $tx^{T'}_{spl}$ with different transactions (order) T' but the same set S_T and digest D_T. There are two cases: 1) If the merge transaction tx^T_{mer} of tx^T_{spl} is on-chain, $tx^{T'}_{spl}$ cannot be committed. Because states in S_T have been updated, resulting in invalidity to split the same digest D_T from available L1 state digest D_{avl}. 2) If tx^T_{mer} is not on-chain yet, $tx^{T'}_{spl}$ is also invalid. Because D_T has been split from D_{avl} and not merged back yet.

For the Non-deterministic Layer-1 Consensus (**e.g.*****, PoW)***, $tx^{T'}_{spl}$ can exist in a fork block with the same block height as the block containing tx^T_{spl}. Thus, the rollback attack cannot be strictly prevented [13]. While, L2chain makes the attacker harder to perform such an attack. Because each execution attempt requires the attacker to generate a valid block containing a split transaction. Thus, L2chain aligns rollback attack difficulty with possessing voting resources in a non-deterministic consensus. For states with many possible values, a rollback attack is as hard as a 51% attack.

Liveness Analysis. We can view liveness issues from the abnormality in two granularity. 1) *Abnormal executors:* $< \frac{1}{2}$ executors in a DApp is abnormal (*e.g.*, slow, deny-of-service or perform maliciously). As TEEs strictly prevent invalid state and digest updates, they can tolerate crash or Byzantine failures of $< \frac{1}{2}$ executors in each DApp. Thus, such a case cannot throttle the system. 2) *Abnormal DApps:* $\geq \frac{1}{2}$ executors in a DApp is abnormal. In security analysis, we demonstrate the hardness of a malicious DApp to subvert the system. However, without any restriction, an abnormal DApp, (deliberately) delaying the state update and digest merge after splitting a digest from the L1 chain raises the liveness issue. Thus, one possible solution is to set a dynamic parameter λ, stipulating that the digest in a split transaction will be invalid (automatically merged back) after λ blocks. Such a λ can be dynamically changed based on the expected processing speed of DApps. Moreover, an L2 incentive mechanism can also help promote decentralized executors working for a DApp, which is orthogonal to the L2 architecture.

7 EXPERIMENTAL STUDY

In this section, we evaluate L2chain by comparing with the following baselines in both permissioned and permissionless scenarios:

- **Quorum:** a variant of Ethereum with confidentiality by encoding and making the private transactions only visible to related nodes. Meanwhile, it supports pluggable consensus protocols with both Raft and IBFT for a permissioned setting and Ethash (the Ethereum PoW protocol) for a permissionless setting.
- **CAPER-based:** a cross-DApp permissioned blockchain. We compare its hierarchical global consensus protocol [4] where each DApp maintains the ledger in its own view and process private transactions internally by local consensus protocols. Meanwhile, the global consensus protocol ensures the global view consistency and deals with cross-app transactions.
- **SlimChain:** an off-chain parallel transaction processing system where off-chain proposers execute each transaction in TEEs to update the state digest and on-chain validators aggregate results into blocks.

7.1 Experiment Settings

DApp and workloads. We use the *KVStore* smart contract of the BLOCKBENCH [24] macro benchmark as the workload logic of each DApp. Specifically, it reads/writes the key-value pairs of system states. We use the YCSB workload [3] with different percentage of read/write transactions (*i.e.*, read-write with 50% read and write transactions, read-only and write-only workloads). Meanwhile, to capture the cross-app transactions, we first uniformly assign state addresses in the workload to each DApp. Then we randomly group two read/write operations to form one transaction such that it either read/write states within one DApp or across two DApps.

System nodes. We control system nodes by the DApp count. Specifically, each DApp has four nodes, referring to executors in L2chain, proposers (also acting as storage nodes) in SlimChain, and chain nodes, maintaining the ledger in CAPER and Quorum. For a fair comparison, we set validators in L2chain and SlimChain with the same number of DApp nodes. Then, we connect one client to each DApp to send workloads. In addition, to evaluate L2chain's fault-tolerant ability compared with a pure L1 solution, at the beginning of each block interval, we manually control each node in Quorum and L2chain to be temporally crash with the probability of ϕ.

Consensus protocol. To capture a permissioned setting, we use Raft consensus protocol to 1) maintain the L1 blockchain ledger in Quorum and L2chain. 2) achieve local consensus among DApp nodes of CAPER and L2chain executors. Meanwhile, to capture a permissionless setting, we use PoW to maintain the blockchain ledger of Quorum, SlimChain and L2chain while remaining Raft as the protocol in L2 network of L2chain.

Witness cache. Each L2chain DApp executor maintains the witness cache proposed in Sec. 5. We vary the parameter of cache group count τ according to Table 2 to evaluate the performance in different settings. Besides, since the witness group is independent from each other, we use multi-thread (16 by default) to accelerate the witness generation and witness cache update after processing each L1 block.

Other parameters. As pointed out in [8], block time, indicating how often to form a new block, does not affect the throughput but higher time leads to higher latency. Thus, we adopt the default time: 1s for Raft and 10s for PoW. Similarly, for on-chain block size and L2chain batch capacity, larger capacity leads to higher throughput and latency. Thus, we fix block size and L2 batch capacity to 256, approximating the average transaction count in Ethereum [2]. In addition, we fix the overall state count to 2^{26} based on unique Ethereum addresses [2] and make them available to every DApps.

Table 2: System Parameters

Parameters	Values
Number of DApps	**4**, 8, 16, 32
Percentage of cross-app transactions	0%, **20%**, 80%, 100%
Number of witness cache group τ	256, **512**, 1024, 2048
Threads to maintain witness caches	4, 8, **16**, 32
DApp node failure rate ϕ	**0%**, 10%, 20%, 30%

Metrics and environments. We vary parameters in Table 2 with default values in bold to evaluate system throughput and latency. Precisely, throughput measures the number of transactions that can be finalized on the main blockchain (or the global consensus is achieved in CAPER) in a unit of time. Meanwhile, latency measures the breakdown average transaction processing time, including transaction execution, consensus (with network message propagation time), and overhead cost which includes transaction encryption and decryption in Quorum and SlimChain and RSA accumulator operations (*e.g.*, membership witness generation/verification and witness cache update) in L2chain. Note that we eliminate the waiting time for transactions to be processed at the DApp node side as we measure the peak throughput by making the system saturated, inevitably leading to a long waiting time.

We conduct the experiments on Microsoft Azure with the Standard_DC16s_v3[3] machines with 1500 Mbps network bandwidth.

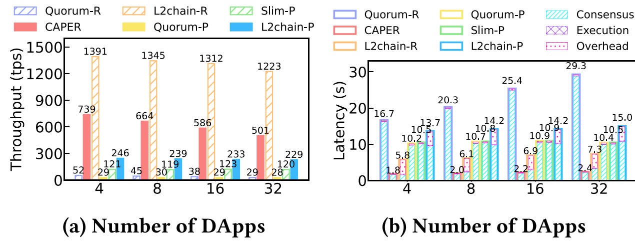
7.2 Experimental Results

Overall Performance. Figures 8, 9 and 10 show the experimental results where L2chain can always achieve the highest throughput. In a permissioned setting, compared with CAPER and Raft Quorum (*Quorum-R*), Raft L2chain (*L2chain-R*) improves the throughput by 1.5X to 2.6X and 21.9X to 42.2X, respectively. In a permissionless setting, PoW L2chain (*L2chain-P*) can achieve 7.1X to 8.9X and 1.7X to 2.2X higher throughput than PoW Quorum (*Quorum-P*) and SlimChain (*Slim-P*). For the latency, L2chain can save 63% to 75% processing time compared with Raft Quorum. However, due to the overhead cost of RSA accumulator and cross-DApp consensus, L2chain exposes higher latency of 2.7X to 3.2X than CAPER, 1.2X to 1.6X than PoW Quorum, and 1.3X to 1.7X than PoW SlimChain. Moreover, as shown in Fig. 11b the overhead cost can be reduced by the multi-thread technique. In addition, we also evaluate L2chain's fault tolerance ability by varying the DApp node failure rate and discuss the trade-offs of L2chain versus an L1 solution. Due to a limited space, we refer more details in our technical report [63].

Impact of the DApps count. As shown in Fig. 8, the throughput of all permissioned systems decreases with more DApps. Meanwhile, the throughput improvement of L2chain increases from 26.8X to 42.2X compared with Raft Quorum, 1.9X to 2.4X compared with CAPER. While, the latency of L2chain decreases from 75% to 65% compared with Raft Quorum, 3.2X to 3.0X overhead compared with

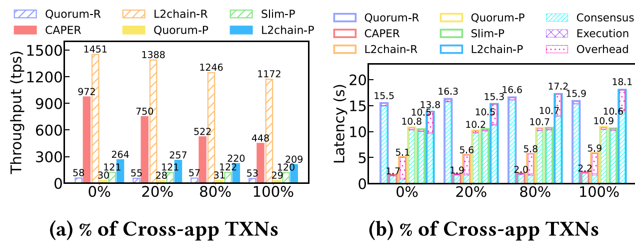
[3]https://docs.microsoft.com/en-us/azure/virtual-machines/dcv3-series

CAPER. It implies L2chain has better scalability in the permissioned setting. However, the throughput improvement of L2chain compared with PoW Quorum and SlimChain does not have a notable change. Because the main bottleneck is still the L1 chain PoW protocol whose performance is not affected by the DApp count.



(a) Number of DApps (b) Number of DApps

Figure 8: Varying the number of DApps



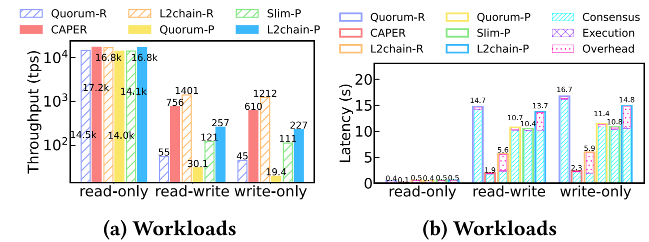
(a) % of Cross-app TXNs (b) % of Cross-app TXNs

Figure 9: Varying the percentage of cross-app transactions

Impact of the cross-app transaction percentage. Because CAPER and L2chain process cross-app transactions by reaching a local consensus with Raft among DApps, as shown in Fig. 9, with a larger cross-app transaction percentage, they have lower throughput and higher latency due to the raising local consensus costs. However, PoW Quorum and SlimChain do not distinguish whether a transaction is cross-app or not. They directly update and record the on-chain state once. Moreover, compared with CAPER, the throughput improvement of L2chain increases from 1.5X to 2.6X. Because every cross-app transaction in CAPER needs to reach a global consensus once, while L2chain processes DApp transactions in batches and only achieves global consensus twice (to record split and merge transactions on the L1 chain) for each batch.

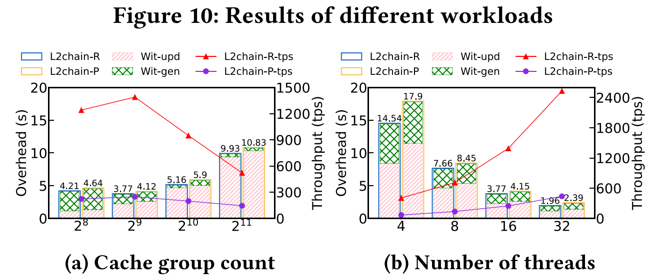
Impact of the workload. Fig. 10 varies the percentage of read-/write operations in the workload. For read-only workload, all systems can achieve over 14k tps since there is no consensus cost. With more write transactions, all system throughput decreases. For the processing latency, L2chain has a similar consensus cost to CAPER, PoW-based Quorum and SlimChain. However, L2chain has 10X more overhead cost than Quorum and 8X than SlimChain. The difference is mainly brought by the usage of the RSA accumulator in L2chain. It also shows the necessity of the proposed witness cache mechanism in L2chain to reduce the overhead cost further.

Impact of witness cache group and processing threads counts. Fig. 11 shows the L2chain overhead cost, consisting of RSA accumulator witness generation (*Wit-gen*) and witness cache update (*Wit-upd*) to process layer-1 split or merge transactions. Specifically, with more cache groups, the cache update cost increases while the witness generation cost decreases. Because each witness cache represents fewer states, reducing the exponential operations to generate a membership witness for each individual state. As analyzed in Sec. 5.2, the optimal cache group count exists (2^9 in our experiments) when achieving the balance between two costs. Meanwhile, as shown in previous experiments, in a permissioned setting, most of the L2chain latency comes from the overhead cost, which can affect the throughput dramatically. In contrast, the consensus cost is still the bottleneck in a permissionless setting. In addition, since each witness cache is independent, as shown in Fig. 11b, increasing the processing thread can reduce the overhead cost near-linearly.



(a) Workloads (b) Workloads

Figure 10: Results of different workloads



(a) Cache group count (b) Number of threads

Figure 11: Varying cache group count and processing thread

8 CONCLUSIONS

In this paper, we design a novel layer-2 blockchain framework namely L2chain to scale the decentralized applications with the guarantees of transaction execution confidentiality and order correctness. We leverage the RSA accumulator and TEE to propose the *split-execute-merge* workflow and *two-step execution* to process transactions. Specifically, L2chain maintains the global system states digest on a layer-1 chain and let layer-2 DApp executors to split a partial digest of a state subset, then privately execute transactions in batches to update the digest. We also design the witness cache and its organization optimization to further reduce the latency. Extensive experiment results show that L2chain can achieve 1.5X to 42.2X and 7.1X to 8.9X throughput improvements in permissioned and permissionless settings respectively.

ACKNOWLEDGMENTS

Lei CHEN is supported by the National Key Research and Development Program of China (2022YFE0200500), National Key Research and Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16213620, CRF Project C6030-18G, C1031-18G, C5026-18G, AOE Project AoE/E-603/18, RIF Project R6020-19, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants PRP/004/22FX , Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab grants and HKUST Global Strategic Partnership Fund (2021 SJTU-HKUST).

REFERENCES

[1] 2018. Istanbul BFT. https://github.com/ethereum/EIPs/issues/650.

[2] 2022. Ethereum Charts and Statistics. https://etherscan.io/charts.

[3] 2022. YCSB. https://github.com/brianfrankcooper/YCSB.

[4] 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.

[5] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. ACM New York, NY, USA.

[6] 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.

[7] Nathan Aw. 2018. Private Data Collections: A High-Level Overview. https://www.hyperledger.org/blog/2018/10/23/private-data-collections-a-high-level-overview.

[8] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. 2018. Performance evaluation of the quorum blockchain platform. *arXiv preprint arXiv:1809.03421* (2018).

[9] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2017. Consensus in the age of blockchains. *arXiv preprint arXiv:1711.03936* (2017).

[10] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*. 222–237.

[11] 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.

[12] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[13] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. 2018. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *arXiv preprint arXiv:1805.08541* (2018).

[14] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. 2019. Trusted computing meets blockchain: Rollback attacks and a solution for hyperledger fabric. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 324–32409.

[15] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph.D. Dissertation. University of Guelph.

[16] V. Buterin. 2018. On-Chain Scaling to Potentially 500 TX/SEC Through Mass TX Validation. https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-throughmass-tx-validation/3477.

[17] Jan Camenisch and Anna Lysyanskaya. 2002. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual international cryptology conference*. Springer, 61–76.

[18] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[19] J. P. M. Chase. 2018. Quorum: A permissioned implementation of ethereum. https://github.com/jpmorganchase/quorum.

[20] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 189–204.

[21] 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.

[22] Christian Decker and Roger Wattenhofer. 2015. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*. Springer, 3–18.

[23] 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.

[24] 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.

[25] Stefan Dziembowski, Lisa Eckey, Sebastian Faust, and Daniel Malinowski. 2017. PERUN: Virtual Payment Channels over Cryptographic Currencies. *IACR Cryptol. ePrint Arch.* 2017 (2017), 635.

[26] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. 2019. BlockchainDB: A shared database on blockchains. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1597–1609.

[27] K. Floersch. 2019. Ethereum Smart Contracts in L2: Optimistic Rollup. https://medium.com/plasma-group/ethereum-smartcontracts-in-l2-optimistic-rollup-2c1cef2ec537.

[28] Alex Gluchowski. 2019. Zk rollup: scaling with zero-knowledge proofs. https://pandax-statics.oss-cn-shenzhen.aliyuncs.com/statics/1221233526992813.pdf.

[29] G. Greenspan. 2015. Multichain private blockchain-white paper. http://www.multichain.com/download/MultiChain-White-Paper.pdf.

[30] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. 2020. Sok: Layer-two blockchain protocols. In *International Conference on Financial Cryptography and Data Security*. Springer, 201–226.

[31] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. Sbft: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.

[32] Siyuan Han, Zihuan Xu, Yuxiang Zeng, and Lei Chen. 2019. Fluid: A blockchain based framework for crowdsourcing. In *Proceedings of the 2019 international conference on management of data*. 1921–1924.

[33] Mike Hearn. 2013. Micro-payment channels implementation now in bitcoinj.

[34] Maurice Herlihy. 2018. Atomic cross-chain swaps. In *Proceedings of the 2018 ACM symposium on principles of distributed computing*. 245–254.

[35] Maurice Herlihy, Barbara Liskov, and Liuba Shrira. 2021. Cross-chain deals and adversarial commerce. *The VLDB Journal* (2021), 1–19.

[36] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA* 11, 10.1145 (2013), 2487726–2488370.

[37] Rami Khalil, Alexei Zamyatin, Guillaume Felley, Pedro Moreno-Sanchez, and Arthur Gervais. 2018. Commit-chains: Secure, scalable off-chain payments. *Cryptology ePrint Archive, Report 2018/642* (2018).

[38] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[39] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems.. In *NSDI*, Vol. 9. 1–14.

[40] Jiangtao Li, Ninghui Li, and Rui Xue. 2007. Universal accumulators with efficient nonmembership proofs. In *International Conference on Applied Cryptography and Network Security*. Springer, 253–269.

[41] Ming Li, Jian Weng, Anjia Yang, Wei Lu, Yue Zhang, Lin Hou, Jia-Nan Liu, Yang Xiang, and Robert H Deng. 2018. Crowdbc: A blockchain-based decentralized framework for crowdsourcing. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (2018), 1251–1266.

[42] 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.

[43] Yunlong Lu, Xiaohong Huang, Yueyue Dai, Sabita Maharjan, and Yan Zhang. 2019. Blockchain and federated learning for privacy-preserved data sharing in industrial IoT. *IEEE Transactions on Industrial Informatics* 16, 6 (2019), 4177–4186.

[44] Ben Lynn. 2007. *On the implementation of pairing-based cryptosystems*. Ph.D. Dissertation. Stanford University Stanford.

[45] matter labs. 2019. zkSync: scaling and privacy engine for Ethereum. https://zksync.io.

[46] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).

[47] Matthias Mettler. 2016. Blockchain technology in healthcare: The revolution starts here. In *2016 IEEE 18th international conference on e-health networking, applications and services (Healthcom)*. IEEE, 1–3.

[48] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. 2017. Sprites: Payment channels that go faster than lightning. *CoRR, abs/1702.05812* (2017).

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

[50] Raiden network homepage. 2019. Raiden network.

[51] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

[52] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. 2020. Scaling verifiable computation using efficient set accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*. 2075–2092.

[53] Sandro Pinto and Nuno Santos. 2019. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.

[54] Joseph Poon and Vitalik Buterin. 2017. Plasma: Scalable autonomous smart contracts. *White paper* (2017), 1–47.

[55] Joseph Poon and Thaddeus Dryja. 2016. The bitcoin lightning network: Scalable off-chain instant payments.

[56] I. Grigg R. G. Brown, J. Carlyle and M. Hearn. 2016. Corda: An introduction. https://docs.corda.net/en/pdf/corda-introductory-whitepaper.pdf.

[57] 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.

[58] Adi Shamir. 1983. On the generation of cryptographically strong pseudorandom sequences. *ACM Transactions on Computer Systems (TOCS)* 1, 1 (1983), 38–44.

[59] Philip Treleaven, Richard Gendal Brown, and Danny Yang. 2017. Blockchain technology in finance. *Computer* 50, 9 (2017), 14–17.

[60] Benjamin Wesolowski. 2019. Efficient verifiable delay functions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 379–407.

[61] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[62] 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.

[63] Zihuan Xu and Lei Chen. 2022. L2chain, Towards High-performance, Confidential and Secure Layer-2 Blockchain Solution for Decentralized Applications (Technical Report). https://github.com/xzhflying/L2chain/blob/main/technical_report.pdf.

[64] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2018. HotStuff: BFT consensus in the lens of blockchain. *arXiv preprint arXiv:1803.05069* (2018).