Front-running Attack in Sharded Blockchains and Fair Cross-shard Consensus

Jianting Zhang*, Wuhui Chen†‡, Sifu Luo†, Tiantian Gong*, Zicong Hong§, and Aniket Kate*¶
*Purdue University,†Sun Yat-sen University, ‡Pengcheng Laboratory
§The Hong Kong Polytechnic University, ¶Supra Research

***Abstract*—Sharding is a prominent technique for scaling blockchains. By dividing the network into smaller components known as shards, a sharded blockchain can process transactions in parallel without introducing inconsistencies through the coordination of intra-shard and cross-shard consensus protocols. However, we observe a critical security issue with sharded systems: transaction ordering manipulations can occur when coordinating intra-shard and cross-shard consensus protocols, leaving the system vulnerable to attack. Specifically, we identify a novel security issue known as finalization fairness, which can be exploited through a front-running attack. This attack allows an attacker to manipulate the execution order of transactions, even if the victim's transaction has already been processed and added to the blockchain by a fair intra-shard consensus.**

To address the issue, we offer Haechi, a novel cross-shard protocol that is immune to front-running attacks. Haechi introduces an ordering phase between transaction processing and execution, ensuring that the execution order of transactions is the same as the processing order and achieving finalization fairness. To accommodate different consensus speeds among shards, Haechi incorporates a finalization fairness algorithm to achieve a globally fair order with minimal performance loss. By providing a global order, Haechi ensures strong consistency among shards, enabling better parallelism in handling conflicting transactions across shards. These features make Haechi a promising solution for supporting popular smart contracts in the real world. To evaluate Haechi's performance and effectiveness in preventing the attack, we implemented the protocol using Tendermint and conducted extensive experiments on a geo-distributed AWS environment. Our results demonstrate that Haechi can effectively prevent the presented front-running attack with little performance sacrifice compared to existing cross-shard consensus protocols.

I. Introduction

Sharding is a prominent approach to enhance blockchain scalability and has been researched extensively by academia [1]–[7] and deployed in the industry [8]–[13]. Its main idea is to horizontally partition the entire blockchain into multiple self-maintaining shards. With more nodes participating in consensus, sharding can assign more shards to process transactions in parallel, bringing higher throughput. However, there is no free lunch here: along with an intra-shard consensus protocol that can be instantiated with any standard Byzantine Fault Tolerant (BFT) state machine replication (SMR) protocols [14], [15], sharded blockchains also need a cross-shard consensus protocol that is specifically designed to handle transactions involving entities and data from two or more shards. The key to a cross-shard consensus protocol is to ensure the isolation property that there is no inconsistent access to the same data and the atomicity property that all relevant shards either commit or abort the cross-shard transaction.

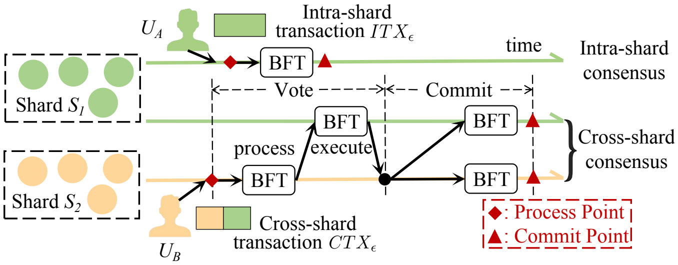


Fig. 1. Finalization unfairness: Assume shard S_1 maintains a contract ϵ. The intra-shard transaction ITX_ϵ is executed and committed before the cross-shard transaction CTX_ϵ even though CTX_ϵ is processed earlier than ITX_ϵ.

Most previous sharded systems adopt a two-phase (2P) protocol to handle cross-shard transactions, motivated by the two-phase commit protocol of traditional distributed databases [16]. The 2P cross-shard protocol works as follows: First, in the vote phase, all shards relevant to the cross-shard transaction modify and lock local states by running one instance of the intra-shard consensus; then, in the commit phase, another instance of intra-shard consensus is used to commit the transaction. For example, as shown in Fig. 1, a user U_B in shard S_2 constructs a cross-shard contract transaction CTX_ϵ to call a smart contract ϵ maintained by shard S_1. The sender's shard (i.e., S_2) first *processes* CTX_ϵ, e.g., deducting U_B's balance used for this transaction. Then, the contract's shard (i.e., S_1) *executes* the contract ϵ with the data generated by S_2 and modifies ϵ's state data. CTX_ϵ will be committed if these two data modifications are operated successfully.

An observation is that transaction *processing* and contract *execution* for a cross-shard contract transaction are completed by different shards and introduce multiple instances of intra-shard consensus. *A cross-shard transaction will thus be packed into many blocks from multiple shards, involving multiple transaction orders.* This is inevitable due to the separation of data in a sharded system. In contrast, these operations for an intra-shard transaction are completed by only one shard via one instance of intra-shard consensus. Therefore, there exists a *processing-execution difference* between intra-shard

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23197
www.ndss-symposium.org

and cross-shard contract transactions. Such a difference is not an issue if the called contract is agnostic to transaction order, e.g., a transfer contract that previous sharded systems focus on [17]–[20]. Unfortunately, there are also contracts where the transaction order is crucial, such as decentralized finance (DeFi) [21], initial coin offerings [22], and gambling [23]. For instance, recently Daian et al. [24] and Qin et al. [25] show that the adversarial manipulation of transaction order in Ethereum DeFi can extract millions of USD from users. In this work, we call such contracts *order-sensitive contracts (OSCs)* and the problem relevant to the processing-execution difference when calling OSCs as the finalization fairness problem. Fairness is a loaded term in the blockchain field, and we formally define finalization fairness for a blockchain sharded system in § IV-C. Informally, it means that transactions[1] are executed and committed in the order they are processed.

We formulate and present the *front-running attack* triggered by this fairness problem (§ III). To the best of our knowledge, this is the first front-running attack in sharded systems. This attack allows the adversary to manipulate the transaction order to have their transactions executed and finalized before the victims' transactions. For instance, an attacker strategically constructs a transaction (e.g., an intra-shard transaction ITX_ϵ) to target a cross-shard transaction (e.g., CTX_ϵ) created by the victim. Since the processing and execution of cross-shard transactions are separated, the attacker's transaction ITX_ϵ can be inserted between the processing and execution of the victim's transaction CTX_ϵ. This will lead that ITX_ϵ is finally executed before CTX_ϵ even though CTX_ϵ is processed and stored on the blockchain earlier than ITX_ϵ (as shown in Fig. 1). Such a front-running attack is inexpensive without transaction fee bidding contests [21], [24], but more likely to succeed in sharded systems because of the existence of the processing-execution difference. As we discuss in § III-B, the front-running attack can not only threaten the security of the above OSCs, but also lead to unbalanced shards, eventually sacrificing the scalability of sharded systems.

To prevent this front-running attack in sharded systems, we propose Haechi[2], a novel cross-shard protocol (§ V). Haechi is a *process-order-execute-commit* (POEC) consensus model whose key idea is to introduce a globally fair ordering phase between transaction processing and execution phases. With the ordering phase, Haechi ensures that the execution order is consistent with the processing order of OSC transactions, thus providing the finalization fairness for sharded systems. In addition to satisfying the finalization fairness, a globally fair order achieves another desirable feature for a sharded system. Specifically, since a global order ensures the consistency of transaction commitment, Haechi supports non-blocking processing for transactions, i.e., conflicting transactions that call the same contracts can be handled continuously in parallel even though some of them are not committed yet. Since in a practical scenario, there are some popular smart contracts that are called by users frequently, the feature of non-blocking processing makes Haechi more practical and promising in supporting real-world smart contracts.

However, since shards have various consensus speeds in a sharded system, establishing a globally fair order for a sharded system faces new challenges. *The challenges come from the in-flight transactions.* Specifically, due to the various consensus speeds, committing a block in the processing phase takes longer for a sluggish shard than it does for a prompt shard. As a result, when transactions in the prompt block are ordered, those in the sluggish block are still being processed and will skip the ordering phase. These *in-flight transactions* of the sluggish block, even if being processed earlier, are ordered after the transactions of the prompt block, compromising finalization fairness. To eliminate in-flight transactions, a straightforward solution is to *synchronize* all shards block-by-block, i.e., any shard is not allowed to propose new blocks until all shards complete consensus for their current blocks, by which all shards have the same block heights and thus there are no in-flight transactions on any block height. However, such a synchronization mechanism is low-efficiency and far from the practicality of sharded systems.

To solve the challenges, Haechi adopts a *finalization fairness algorithm* to coordinate a globally fair order for a sharded system. Instead of asking all shards to synchronize their block heights, the algorithm allows each shard to work independently without being affected by the ordering phase or other shards, while yet guaranteeing finalization fairness. To achieve this, Haechi first uses the *block timestamp* as the ordering indicator. Since the block timestamp can represent the time when transactions are processed and is incremental in each shard, it can be used to establish a *deterministic global order* in which transactions that are processed first must be ordered first. Furthermore, since the block timestamp is publicly verifiable and immutable once a block is committed, Haechi prevents the adversary from manipulating the order by forging the block timestamp. Then, considering in-flight transactions, Haechi adopts a so-called *at-least-one* ordering rule. Specifically, we observe that our protocol can catch enough information and start an ordering phase without waiting to receive all in-flight transactions as long as it receives one block from each shard. The finalization fairness algorithm allows Haechi to reduce performance loss caused by the global ordering phase. We present security analysis in § VII.

Overall, this work makes the following contributions:

- We formulate and present the front-running attack against the current 2P cross-shard consensus protocol, which, to the best of our knowledge, is the first front-running attack in sharded systems. This front-running attack can be devastating: (i) shards receiving unbalancing transaction requests, sacrificing the scalability of sharded systems; (ii) OSCs built on sharded systems pose a significant security risk. These unfavorable outcomes further limit applications of sharding technology.
- We give a new fairness definition for sharded systems, called finalization fairness, which demands that any two transactions calling the same OSC must be executed in the same order as they are processed. It is a requisite for preventing front-running in sharded systems.
- We design Haechi, a novel cross-shard protocol with two improvements: (i) it provides finalization fairness and prevents the front-running attack in a sharded system; (ii) it enhances the efficiency of handling OSC transactions.
- We implement Haechi with Tendermint [14] and compare it with prior state-of-the-art cross-shard consensus protocols.

[1] In the rest of the paper, transactions represent contract transactions, and transaction executions represent executions of the called contracts.

[2] Haechi is a mythical creature representing justice and fairness.

We evaluate them by running up to 990 nodes on geo-distributed AWS EC2 instances spread across ten regions, showing that Haechi can achieve 13,000+ TPS, less than 6s intra-shard confirmation latency, and less than 17s cross-shard confirmation latency with 33 shards. We also evaluate the presented cross-shard front-running attacks for different cross-shard protocols, showing that Haechi can effectively prevent such attacks while existing 2P cross-shard protocols are vulnerable to them.

II. Preliminaries

In this section, we start with an example of order-sensitive contracts (OSCs) in a blockchain. Then, we give a concrete working process on how the classic 2P cross-shard consensus protocol handles a cross-shard OSC transaction.

A. An Example of Order-sensitive Contract

OSCs are a type of smart contract whose states are relevant to transactions order. For instance, if transactions TX_1 and TX_2 call an OSC, executing TX_1, TX_2 and executing TX_2, TX_1 will result in different OSC states. For illustration purposes, we give an example of OSC that is derived and simplified from Uniswap [26], one popular decentralized exchange contract. This OSC allows users to buy *tokens* with their *coins*, where tokens are the data of contract state and coins are the data of user accounts[3].

TokensExchange Contract. Fig. 2 presents our example, TokensExchange, written in Solidity [27]. TokensExchange consists of: (i) a state variable *factory*[4] that records core information of the current exchanging market, like tokens pairs, price, and liquidity (Line 3); (ii) a contract function *swapCoinForTokens* realized to swap a specific token with the user's coins (Line 5). This function has three parameters (i.e., input data). The parameter *path* specifies multiple intermediate Token addresses used for this token exchange. The parameter *to* represents the address of the Token that the user is swapping. The parameter *msg.value* represents the number of coins that the user pays for this exchange. The whole token exchanging process is as follows. First, the contract computes how many actual tokens the user can get with the input coins, i.e., computes the token price (Line 8). Then, the contract transfers the coins into a wrapper token existing in the *factory* (Line 9). For example, in Ethereum, the coin *Ether* will be wrapped into a type token in the *factory*, called *WETH*. This operation is used to provide a more secure and convenient way for token exchanges. Finally, in Lines 10-15, the exchanging market updates the price and liquidity of the involved tokens in the routing path, and the user receives the fitting amount of tokens for its exchanged coins.

According to the function realized in TokensExchange, if a contract transaction is committed successfully, then the user will lose its coins for obtaining relevant tokens, and the contract will update its contract state.

```
1  contract TokensExchange {
2    // state variables of the contract
3    address public factory;
4    // method to exchange a token with coin
5    function swapCoinForTokens(address[] calldata path
         , address to, uint msg.value)
6    {
7      // msg.value is the user's coins
8      amounts = getAmountsOut(msg.value, path);
9      transferCoin(factory, amounts[0]);
10     for (uint i; i < path.length - 2; i++ {
11       address _to = factory.getTokensAdd(path[i+1]);
12       swapAndUpdate(factory, path[i], path[i+1],
             amount[i+1], _to);
13     }
14     swapAndUpdate(factory, path[path.length-2], path
           [path.length-1], amount[path.length-2], to);
15   }
16   // other state variables and methods...
17 }
```

Fig. 2. The code of a simple decentralized exchange contract.

B. 2P Cross-shard Protocol in Sharded Systems

Now, consider the implementation of such an OSC in a sharded system. We assume a user in shard S_1 constructs a cross-shard transaction CTX_ϵ to call TokensExchange contract (denoted as ϵ) maintained by shard S_2. In this example, CTX_ϵ will modify the data of two accounts in two operations. The first, i.e., *transaction processing*, is that CTX_ϵ withdraws coins (i.e., the *msg.value*) of the user's account in S_1. The second, i.e., *contract execution*, is that CTX_ϵ modifies the contract state of ϵ (i.e., the state variable *factory*) in S_2, including transferring tokens to the user, updating token price and liquidity. We emphasize that in a scenario of calling contracts, the later operations are invoked only after the former operations are completed. Because the later operations rely on data generated after the former operations and contract interactions are unpredictable [28], [29], e.g., a contract invokes other different contracts based on the block height when it is executed, but the block height is uncertain.

Fig. 3 shows the process of handling CTX_ϵ with the 2P cross-shard protocol in a sharded system. Similar to traditional databases, the 2P cross-shard protocol is also driven by a coordinator, such as a client [2], [30], or a shard [3], [17], [18], [31]. In this example, we assume the coordinator is the shard that manages the account of the transaction's sender, i.e., S_1. The 2P cross-shard consensus protocol contains two phases: the vote phase and the commit phase.

Phase 1: Vote for the processing and execution results. In this vote phase, the coordinator S_1 leads messages delivery among all related shards for collaboratively handling CTX_ϵ. The specific process can be concluded as follows:

Step ❶: S_1 processes CTX_ϵ, where it will modify the data of the transaction's sender (i.e., the user's coins). Specifically, S_1 checks if the user has sufficient coins to pay for the transaction (i.e., *msg.value* used for exchange), and locally withdraws these coins from the user's account. Then, CTX_ϵ is packed into a new block SC_1^i by a round of intra-shard consensus, and the relevant coins of the user will be locked for paying until the cross-shard consensus is completed.

Step ❷: S_1 creates a message used for contract execution. Specifically, S_1 packs the input data (i.e., function parameters *path, to*, and *msg.value* in this example), the ϵ's address, and a commitment quorum certificate on the block SC_1^i, into a relay

[3] See differences between tokens and coins on https://blog.liquid.com/coin-vs-token.

[4] For illustration, we assume the token pools (i.e., UniswapPair contracts) are a part of the state of this Router contract, recorded in the variable *factory*.

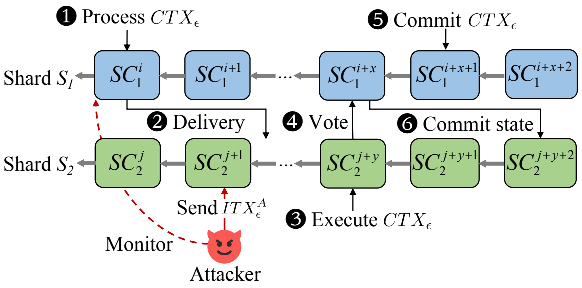


Fig. 3. The process of 2P cross-shard consensus protocol with a shard-based coordinator.

message. This message is used as a proof and request showing that a cross-shard transaction has been processed successfully by the sender's shard. Finally, the relay message is forwarded to S_2 for the contract execution.

Step ❸: The contract shard S_2 executes CTX_ϵ. When receiving the relay message after a network delay (e.g., after y blocks), S_2 verifies its validity. Then, S_2 retrieves ϵ according to its address and executes it with the input data by a round of intra-shard consensus. After that, S_2 locally modifies and locks the changing states of ϵ (i.e., *factory* in this example). The relay message and changing states are recorded into SC_2^{j+y}.

Step ❹: S_2 votes for the commitment. If the execution is successful, S_2 sends the execution results with a *vote-for-acceptance* to S_1. Otherwise, it sends a *vote-for-abort* to S_1.

Phase 2: Commit the transaction and state changes. In the second commit phase, the coordinator S_1 collects all voting results to determine if CTX_ϵ can be committed.

Step ❺-❻: If S_1 receives a vote-for-acceptance from S_2, S_1 unlocks the relevant coins of the user and commits CTX_ϵ into its ledger SC_1 (Step ❺). At the same time, the coordinator S_1 sends a *commit-for-acceptance* to S_2 where S_2 will unlock and commit the changing contract states of ϵ (Step ❻). In contrast, S_1 aborts the commitment of CTX_ϵ and sends a *commit-for-abort* to S_2 for aborting state changes of ϵ.

The processing-execution difference. In this example, we find that handling a cross-shard contract transaction is completed by multiple shards. In particular, transaction processing operation, i.e. Step ❶, is completed by S_1 while transaction execution operation, i.e., Step ❸, is completed by S_2. In contrast, handling an intra-shard contract transaction only involves one shard. For example, assume a user U_B in S_2 calls ϵ by constructing an intra-shard transaction ITX_ϵ^B. Since S_2 maintains both U_B's account and ϵ's account, ITX_ϵ^B can be committed by S_2 independently via only one intra-shard consensus. Therefore, there exists a processing-execution difference between intra-shard and cross-shard transactions. In the next section, we will present how the attacker utilizes such a processing-execution difference to launch a front-running attack.

III. Front-Running in Sharded Systems

A. Attacking Process

Recall from the process of 2P cross-shard consensus in Fig. 3, the called contract ϵ starts to be executed with CTX_ϵ only after CTX_ϵ is processed by S_1 via an intra-shard consensus and is forwarded to S_2 after a period of network latency. Obviously, *there is a time difference between the time when a cross-shard transaction is processed and the time when the called smart contract is actually executed.*

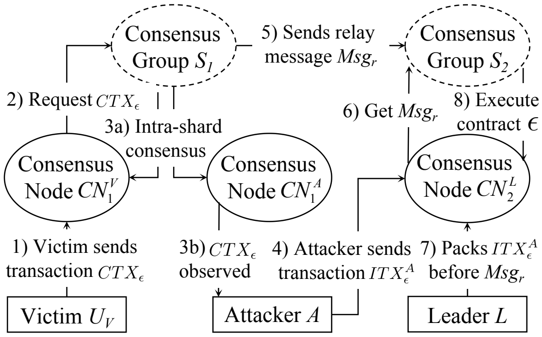

Consensus Group S_1
5) Sends relay message Msg_r
Consensus Group S_2
2) Request CTX_ϵ
3a) Intra-shard consensus
6) Get Msg_r
8) Execute contract ϵ
Consensus Node CN_1^V
Consensus Node CN_1^A
Consensus Node CN_2^L
1) Victim sends transaction CTX_ϵ
3b) CTX_ϵ observed
4) Attacker sends transaction ITX_ϵ^A
7) Packs ITX_ϵ^A before Msg_r
Victim U_V
Attacker A
Leader L

Fig. 4. Intra-shard attacking: the attacker monitors the target shard chain and requests a front-running transaction to be executed before the victim's cross-shard transaction.

Intra-shard attacking model. The attacker can utilize such a time difference to launch a front-running attack against the cross-shard transaction with an intra-shard transaction. Briefly, as transaction processing and execution are separated for a cross-shard transaction, the attacker constructs and has its intra-shard transactions executed between the processing and execution of the victim's cross-shard transaction. Fig. 4 describes the attacking process concretely, and a brief illustration is shown at the bottom of Fig. 3. Specifically, to attack CTX_ϵ, the attacker A first monitors the network of S_1, e.g., it runs a consensus node CN_1^A in S_1. Once the attacker observes CTX_ϵ will appear in the next new block, it creates a front-running intra-shard transaction ITX_ϵ^A with its account managed by S_2 and sends ITX_ϵ^A to the shard leader CN_2^L of S_2. Note that the attacker can observe CTX_ϵ in advance rather than waiting for the new block to be finally appended into the shard chain SC_1. That is because a BFT-based intra-shard consensus always requires several stages before reaching a final agreement, e.g., PBFT [32] contains pre-prepare, prepare, and commit stages, of which the attacker can receive the new block at the pre-prepare stage. Furthermore, since CTX_ϵ is being processed by S_1 (Step ❶) and requires a period of transmission time before arriving at S_2 (Step ❷), ITX_ϵ^A is more likely to be executed by S_2 and changes the contract state of ϵ before CTX_ϵ (i.e., before Step ❸). Thus, although CTX_ϵ is processed by the sharded system earlier than ITX_ϵ^A, ITX_ϵ^A is finally executed and committed earlier than CTX_ϵ. Finally, the victim U_V is front-running attacked by the attacker.

More flexible attacking model. The above attacking model is based on the processing-execution difference between intra-shard and cross-shard transactions. This requires the attacker to create multiple accounts in different shards if it wants to front-run multiple contracts from different shards with its intra-shard transactions. In addition, the attacker can also adopt a more flexible attacking model, called *cross-shard attacking*, by utilizing the processing-execution difference between cross-shard transactions from distinct shards (see appendix A for more details). The idea is that the attacker only creates accounts in a shard with the fastest consensus speed. In this case, the attacker can still front-run the victims' transactions with its cross-shard transactions. Specifically, the shard where the attacker's accounts are located processes transactions faster than other shards, allowing the attacker's cross-shard transactions to reach the target contract's shard before the victims' cross-shard transactions. As a result, the target contract's shard potentially executes the attacker's transactions before the victims' transactions, resulting in successful front-running attacks.

B. Impacts on Sharded Systems

We note that to launch such a front-running attack in a sharded system, the adversary only needs to monitor the network and register accounts in some shards. It neither needs to compromise the shard security nor involves transaction fee bidding contests. Such low-cost attacks will allow the adversary to conduct front-running transactions intensively, bringing devastating impacts to a sharded system.

On the one hand, the front-running attack will compromise the scalability of a sharded system. Specifically, to reduce the risk of being front-run, a user would like to register an account in the contract's shard, by which it can avoid the 2P cross-shard consensus. However, such a strategy leads to workload unbalancing among shards, i.e., distinct shards will manage a various number of accounts and handle a various number of transactions. In a real-world scenario, some popular contracts occupy the major transaction volume of the network. For instance, according to the data provided by Etherscan[5], Uniswap accounts for nearly 11% of daily transactions in the Ethereum network from Nov. 25th, 2022, to Dec. 25th, 2022. The unbalancing workload among different shards will significantly limit the scalability of sharding technology, i.e., even as the number of shards increases, a sharded system cannot handle more transactions because most transactions are created in only several specific shards. On the other hand, the front-running attack also threatens the security of many decentralized applications that are affected by transaction ordering manipulations significantly, such as DeFi and gambling.

It is crucial to prevent such front-running attacks so that a sharded system can be applied to a practical scenario supporting thousands of decentralized applications. However, to the best of our knowledge, none of the current sharded systems consider such issues. This paper thus proposes a novel cross-shard consensus protocol, called $Haechi$, to fill this gap.

IV. Problem Statement

A. System Definitions

We target the setting of state-sharded systems with smart contract capabilities, where the whole ledger is partitioned among several sets of nodes by a Sybil-resistance protocol, e.g., Proof-of-Work (as in [1]–[3]) or Proof-of-Stake (as in [8], [11], [13]). Besides, shards adopt a BFT-based protocol [14], [15], [32] as their intra-shard consensus. There are $(m+1)$ shards in the system, in which a shard $S_i (1 \leq i \leq m)$ maintains a separate shard chain $SC_i = \{SC_i^1, SC_i^2, ...\}$, where SC_i^b is the b-th block in SC_i. Each shard block is attached with a block timestamp to represent the time of block generation. Furthermore, there is a beacon shard S_0 that is responsible for maintaining a *beacon chain* $BC = \{BC^1, BC^2, ...\}$. The beacon chain[6] is widely used by previous sharded systems, which is used to coordinate cross-shard communications [5], [11], [17], [33] or maintain the global configuration for the sharded system [2], [3], [9], [34], such as shard number, node identity, epoch randomness, etc.

B. Trust Assumptions and Adversary Model

We assume there is a reliable and secure shard reconfiguration (or resharding) mechanism like prior sharded systems [2], [3], by which the sharded system adjusts all shards periodically to ensure that the honest nodes always control each shard. Specifically, a shard $S_i (0 \leq i \leq m)$ always satisfies $3f_i + 1 < |S_i|$, where f_i is the number of byzantine nodes and $|S_i|$ is the shard size. In light of this assumption, the following security properties are always guaranteed by the intra-shard BFT protocol in each shard:

- Intra-shard safety. Any two honest nodes of the same shard store the same prefix ledger.
- Intra-shard liveness. If a transaction is received by at least one honest node, then the transaction will be eventually handled and get a response from the shard.

Furthermore, we assume that the adversary can behave arbitrarily to launch attacks to have its transactions executed before the victim's transactions. However, the adversary is computationally bounded.

We assume a partially synchronous network model since each shard independently runs a partially synchronous SMR protocol (i.e., Tendermint BFT [14] in this work). However, we emphasize that Haechi can also work in an asynchronous network since it does not rely on any time-bound. Specifically, if shards run an asynchronous SMR protocol (e.g., Tusk [35]), then the network assumption could be asynchrony.

C. Fairness in Sharded Systems

The front-running attack discussed in § III reflects a fairness issue where the adversary can manipulate the execution order of transactions independent of their processing order. Therefore, to give a clear definition of fairness for sharded systems, we first define two ordering relations for transactions: processing order and execution order.

Definition 1 (Processing order)**.** *Let $\prec_{\mathbf{P}}$ denote a processing order relation in a sharded system, $TX_{\mapsto\epsilon} \in SC_i^j$ denote that transaction $TX_{\mapsto\epsilon}$ calling contract ϵ is recorded in block SC_i^j. Given transactions $TX_{\mapsto\epsilon}^1$ and $TX_{\mapsto\epsilon}^2$, we define $TX_{\mapsto\epsilon}^1 \prec_{\mathbf{P}} TX_{\mapsto\epsilon}^2$ if one of the following three conditions is satisfied:*

(i) $TX_{\mapsto\epsilon}^1, TX_{\mapsto\epsilon}^2 \in SC_i^m$, and $f_{idx}(TX_{\mapsto\epsilon}^1) < f_{idx}(TX_{\mapsto\epsilon}^2)$, where $f_{idx}(\cdot)$ is a function returning the indexes of transactions in a block.

(ii) $TX_{\mapsto\epsilon}^1 \in SC_i^m$, $TX_{\mapsto\epsilon}^2 \in SC_i^n$, and $m < n$, where m and n represent the block height of the same chain SC_i.

(iii) $TX_{\mapsto\epsilon}^1 \in SC_i^m$, $TX_{\mapsto\epsilon}^2 \in SC_j^n$, and $f_{bt}(SC_i^m) < f_{bt}(SC_j^n)$, where $f_{bt}(\cdot)$ is a function returning the timestamp of a block.

[5] https://etherscan.io/stat/dextracker?range=30

[6] In the previous sharded systems, the beacon chain also called the identity chain [2] or reference chain [3], [17], [33], etc.

In a sharded system, two transactions may be recorded in the same block (condition i), different blocks of the same shard (condition ii), and different shard chains (condition iii). Therefore, the processing order $\prec_{\mathbf{P}}$ defines a relation for two transactions based on the time *when transactions are processed*. If $TX^1_{\mapsto\epsilon} \prec_{\mathbf{P}} TX^2_{\mapsto\epsilon}$, we say $TX^1_{\mapsto\epsilon}$ has been processed before $TX^2_{\mapsto\epsilon}$.

Definition 2 (Execution order)**.** *Let $\prec_{\mathbf{E}}$ denote an execution order relation in a sharded system, $TX_{\mapsto\epsilon}$ denote a transaction that calls contract ϵ, $TX_{\mapsto\epsilon,\epsilon_k}$ denote a sub-transaction generated to call contract ϵ_k when executing ϵ, and $\Upsilon(TX_{\mapsto\epsilon})$ be the set of contracts that are called when executing ϵ. Given two transactions $TX^1_{\mapsto\epsilon_1}$ and $TX^2_{\mapsto\epsilon_2}$, we define $TX^1_{\mapsto\epsilon_1} \prec_{\mathbf{E}} TX^2_{\mapsto\epsilon_2}$ if: for every $\epsilon_k \in \Upsilon(TX^1_{\mapsto\epsilon_1}) \cap \Upsilon(TX^2_{\mapsto\epsilon_2})$, sub-transaction $TX^1_{\mapsto\epsilon_1,\epsilon_k}$ is executed before $TX^2_{\mapsto\epsilon_2,\epsilon_k}$.*

The execution order $\prec_{\mathbf{E}}$ defines a relation only for two transactions involving the same contracts, i.e., transactions exist dependencies. For example, we assume that transaction $TX^1_{\mapsto\epsilon_1}$ calls contracts ϵ_1, ϵ_3, and ϵ_4 respectively with sub-transactions $TX^1_{\mapsto\epsilon_1,\epsilon_1}$, $TX^1_{\mapsto\epsilon_1,\epsilon_3}$, and $TX^1_{\mapsto\epsilon_1,\epsilon_4}$. Similarly, another $TX^2_{\mapsto\epsilon_2}$ calls contracts ϵ_2, ϵ_3, and ϵ_4 respectively with sub-transactions $TX^2_{\mapsto\epsilon_2,\epsilon_2}$, $TX^2_{\mapsto\epsilon_2,\epsilon_3}$, and $TX^2_{\mapsto\epsilon_2,\epsilon_4}$. If $TX^1_{\mapsto\epsilon_1,\epsilon_3}$ is executed before $TX^2_{\mapsto\epsilon_2,\epsilon_3}$, and $TX^1_{\mapsto\epsilon_1,\epsilon_4}$ is executed before $TX^2_{\mapsto\epsilon_2,\epsilon_4}$, then we have $TX^1_{\mapsto\epsilon_1} \prec_{\mathbf{E}} TX^2_{\mapsto\epsilon_2}$. With $\prec_{\mathbf{P}}$ and $\prec_{\mathbf{E}}$, we now give a new fairness definition, called finalization fairness, for a sharded system.

Definition 3 (Finalization fairness)**.** *A sharded system is said to possess finalization fairness property on a contract ϵ, if $\epsilon \in \Upsilon(TX^1_{\mapsto\epsilon_1}) \cap \Upsilon(TX^2_{\mapsto\epsilon_2})$ for any given transactions $TX^1_{\mapsto\epsilon_1}$ and $TX^2_{\mapsto\epsilon_2}$, it satisfies: when $TX^1_{\mapsto\epsilon_1} \prec_{\mathbf{P}} TX^2_{\mapsto\epsilon_2}$, then $TX^1_{\mapsto\epsilon_1} \prec_{\mathbf{E}} TX^2_{\mapsto\epsilon_2}$.*

The finalization fairness indicates that if a transaction is processed earlier than another transaction, it should be executed and finalized before the other transaction, i.e., *first-processing first-execution*. Note that not all contracts (e.g., non-order-sensitive contracts) need to be provided finalization fairness, thus a sharded system can be considered to satisfy the finalization fairness as long as it provides such a property for an order-sensitive contract.

D. Goals

We aim to design a cross-shard consensus protocol to prevent a specific front-running attack from happening across shards (i.e., *cross-shard front-running*) as discussed in § III. Note that the attacking behaviors of such a front-running attack are different from previous front-running attacks. The latter happens within a shard (i.e., *intra-shard front-running*) during the period from when transactions appear in the mempool to when transactions are packed into blocks. However, we emphasize that existing solutions of intra-shard front-running are completely compatible with our protocol and can be easily integrated into our protocol (see § X for more discussions).

Obviously, the finalization fairness is the opposite of the front-running attack, where a victim transaction is processed earlier but executed later than a front-running transaction. We next show how Haechi provides finalization fairness property and thus prevents front-running attacks in a sharded system.

V. Haechi Overview

In this section, we present an overview and technical challenges of Haechi, followed by a strawman solution. We emphasize Haechi is designed for order-sensitive contracts (OSCs) that are vulnerable to front-running attacks, such as DeFi [21] and ICO [22]. Therefore, Haechi is implemented as a pluggable cross-shard protocol for the existing sharded systems[7]. For a clearer illustration, we exclusively consider all involved contracts as OSCs in the rest of this paper. Similarly, if not otherwise specified, all the transactions after that are OSC transactions (OTXs).

Observation. Recall that in the front-running attack, an adversary can construct an intra-shard or a cross-shard transaction that is executed earlier than the victim's cross-shard transaction, although the latter is processed first. The main reason is that *no global transaction order is used for intra-shard transactions and cross-shard transactions when they are handled by different shards, respectively*. Motivated by this, Haechi introduces an ordering phase for realizing a globally fair order for OSC transactions.

A. The Process-Order-Execute-Commit Model

Haechi is a process-order-execute-commit (POEC) model that innovatively introduces an ordering phase compared with the 2P cross-shard protocol. We assign the beacon chain to help order transactions in the ordering phase (see more details in § VI-A). The data structure used in communication between the beacon chain and shard chains is *CrossLink*. A CrossLink CL is corresponding to one block of a shard chain, consisting of the following components:

$$CL = < blockTS, L_{tx}, i, h >$$

Here, $blockTS$ is the block timestamp of the corresponding block, L_{tx} is a list of OTXs, i is the shard chain ID, and h is the block height of the corresponding block. The beacon chain works in cycles. In each ordering cycle, the beacon chain picks received CrossLinks and orders OTXs of CrossLinks via one instance of the intra-shard consensus.

Fig. 5 gives an overview of Haechi, which takes the following phases to commit OSC transactions: (i) first, a *processing phase* processes and creates a partial order for transactions, and generates CrossLinks; (ii) then, in an *ordering phase*, the beacon chain decides a total order of transactions in received CrossLinks; (iii) next, in an *execution phase*, the ordered transactions are sent to related shards to execute OSCs; (iv) finally, transactions are committed or aborted by their relevant shards based on the execution results. For distinction, in the following sections, we also name a shard that processes a transaction as the sender shard and a shard that executes the called contract as the contract shard of the transaction.

In a nutshell, Haechi achieves finalization fairness by consistently ordering intra-shard and cross-shard transactions before they execute the same OSC. As a result, Haechi eliminates the processing-execution difference on those transactions. The question now is how to establish a globally fair order in a

[7] In our implementation, we use a variable to represent different types of contracts, and the system chooses the relevant cross-shard protocol to handle transactions based on the type variable.

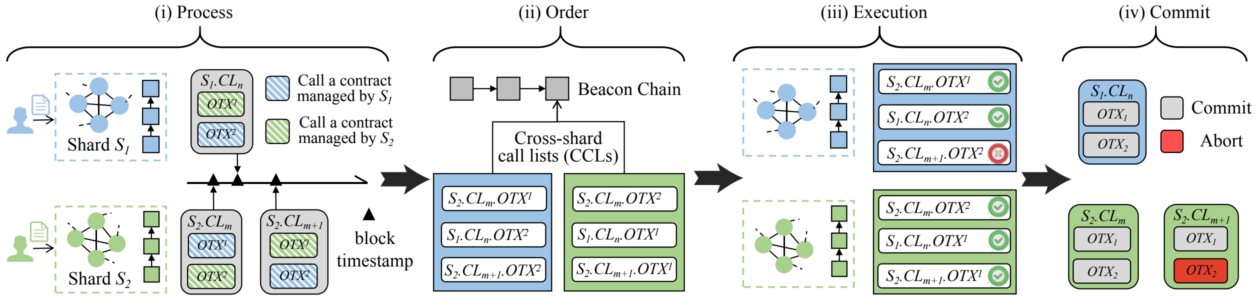


Fig. 5. Haechi overview: (i) the sender shards process transactions; (ii) the beacon shard orders transactions; (iii) the contract shards execute contracts; (iv) transactions are committed in senders' and contracts' shards.

sharded system, i.e., transactions that are processed first must be ordered before those that are processed later, regardless of which shards they come from.

B. Challenges

In a sharded system, however, establishing a globally fair order is non-trivial. A protocol must ensure that all *in-flight transactions* from different shards can be ordered fairly. Specifically, because of the consensus independence and node heterogeneity of shards (e.g., diverse computer power of nodes, network communication, the number of transaction requests, etc.), different shards have different consensus speeds. Thus, it is impossible for all shards to commit transactions synchronously. Furthermore, due to network delays, the beacon chain may receive transactions from various shards in an order different from their processing time. In some cases, there exist some called in-flight transactions that are being processed by shards or delivered to the network, but the beacon chain has not yet received their corresponding CrossLinks. Such in-flight transactions may be processed earlier than those transactions that the beacon chain has received, and thus should be ordered before to ensure the finalization fairness. For instance, consider a scenario in Fig. 6. Note that the message structure between shards and the beacon chain is the CrossLink, labeled with $S_i.CL_j$, where i is the shard ID, and j is the block height of shard S_i.

Illustrative Example: Assume the beacon chain starts to order transactions after receiving $S_1.CL_{n+1}$ from shard S_1 (i.e., at $\triangleleft S_1.CL_{n+1}$ in Fig. 6). Since the beacon chain receives the corresponding CrossLink of SC_2^m at the time $\triangleleft S_2.CL_m$ and $\triangleleft S_2.CL_m > \triangleleft S_1.CL_{n+1}$, the in-flight CrossLink $S_2.CL_m$ cannot be ordered in the same ordering cycle as $S_1.CL_{n+1}$. This will lead to the transactions in SC_1^{n+1} being ordered before the transactions in SC_2^m. However, the transactions in SC_2^m are processed earlier than SC_1^{n+1} because $\triangleright SC_2^m < \triangleright SC_1^{n+1}$. As a result, the finalization fairness is violated.

C. Strawman: A Synchronization Ordering Mechanism

To address the above challenges, an intuitive solution is to eliminate the in-flight transactions by introducing a synchronization ordering mechanism. Specifically, when a prompt shard commits a block (e.g., assume the block height is b) and forwards the relevant CrossLink to the beacon chain, the shard will not propose a new block with height $b+1$ until the beacon chain receives all CrossLinks with the height b from all shards. In other words, the prompt shards need to wait for the sluggish shards before entering the next round of intra-shard consensus. By doing this, all shards synchronize to the same block height regardless of their consensus speeds. As a result, there are no in-flight transactions that will be ignored in each ordering cycle, and all transactions will be ordered fairly based on their processing time no matter which shards they come from.

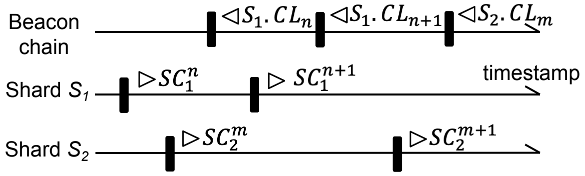


Fig. 6. The consensus difference: shards S_1 and S_2 have distinct consensus speeds, where $\triangleright$ indicates the start time of processing a block and $\triangleleft$ indicates the time when the beacon chain receives the corresponding CrossLink.

Although synchronizing blocks among shards can eliminate potential in-flight transactions and ensure a globally fair order, such a synchronization ordering mechanism is low-efficiency and impractical. In particular, in each ordering phase, only one block's transactions are ordered for each shard. The system performance, therefore, will be limited by the slowest shard, which violates the scalability inherence of sharded systems.

VI. Haechi Details

In this section, we detail Haechi, where we show how Haechi solves the above challenges.

A. Processing Phase

The processing phase of Haechi is comparable to the first step of the vote phase in 2P cross-shard protocols where it involves manipulating the senders' accounts and checking their constraints. This is the phase when OTXs are first handled and packed into blocks. Since Haechi decouples the transaction processing, execution, and commitment, we introduce a new block structure motivated by work [36] for efficiency reasons.

Haechi block structure. Blocks in Haechi consist of two types of transaction lists: *pending list* and *committing list*. The pending list records OTXs that have been processed but have not been committed in the current block height. The committing list is used to exclusively pack transactions that can be committed, including non-OSC transactions (i.e., non-contract transactions or contract transactions that don't invoke OSCs) and OTXs that have been executed successfully and are ready for commitment. Such a two-list block structure benefits the efficiency of Haechi.

First, processing and committing OTXs in distinct lists of blocks enables Haechi to achieve *asynchronous processing*, meaning that OTX processing and execution can be completed asynchronously in different relevant shards. Specifically, with the two-list block structure, OTXs can first be processed and recorded in the pending lists, and the sender shards temporarily hold the data changes. After the OTXs are executed by the contract shards, the committing lists of the later blocks are used to record the execution results of relevant contracts and commit previous pending data changes. This enables asynchronous processing for an OTX where the transaction is processed in the sender shard and is executed in the contract shard asynchronously. The asynchronous processing avoids the sender shards from fetching contract states from the contract shards and executing contracts on their own.

Second, ordering transactions in the pending list before executing them enables Haechi to achieve *non-blocking processing*, meaning that shards can continuously process new OTXs without waiting for the commitments of previous OTXs. The previous 2P cross-shard protocol needs to lock all relevant data to ensure consistent commitments by preventing conflicting transactions from reading/writing the same data in parallel (see appendix B for more details). In this case, a later transaction accessing the same data cannot be processed until the former transaction is committed, leading to blocking processing for these conflicting transactions. In contrast, the two-list block structure enables Haechi to form a local order of a shard for conflicting transactions in the pending list, and this local order will help to finally create a global order among all shards. This global order prevents inconsistent commitments even if there are multiple shards processing conflicting transactions in parallel. As a result, a shard can continuously process new OTXs, and pack more than one transaction calling the same OSC into a shard block in a batching manner.

Algorithm 1 shows the procedure of the processing phase. Once a shard node receives a new block, it goes through all transactions of the blocks (Lines 2-8): (i) the node verifies new OTXs of the pending list by checking if the senders have sufficient balance to pay for transactions (Line 4). Then a valid OTX is added to a transaction list L_{tx}, which is the batching implementation for OTX transmission (Line 6); (ii) the node commits updated states of transactions in the committing list (Line 8, details are in § VI-D). After the new block is handled, the leader of S_i generates a corresponding CrossLink CL_h with the block timestamp $blockTS$, the transaction list L_{tx}, and shard id i, block height h (Line 10). CL_h is broadcast to other nodes in S_i for collecting signatures (Line 13). Once the leader collects a quorum of signatures (i.e., a quorum certificate), it sends CL_h to the beacon chain (Line 11).

B. Ordering Phase

The principal goal of the consensus phase is to create a globally fair order for OTXs received from the processing phase. Haechi uses the beacon chain as the entity to accomplish the ordering phase. In previous sharded systems, a beacon chain has been designed as a global coordinator used for global configuration maintenance [2], [3], [34] or cross-shard coordination [5], [11], [17], [33]. Due to the property of global coordination, Haechi extends the beacon chain to assist in ordering OTXs in its blocks. This makes our solution adaptive and easy to be integrated into existing sharded systems. But we emphasize that any shard in the system can be designated to globally order OTXs.

Algorithm 1 Haechi processing phase (for nodes N_k of shard S_i) in block SC_i^h

```
1: let $\{TX\}_h$ is the set of transactions in $SC_i^h$, $CL_h = \emptyset$ is the
   CrossLink of $SC_i$ in block height $h$
2: for each $tx \in \{TX\}_h$ do,
3:     if $tx$ is a new OSC transaction then
4:         $isValid \leftarrow Processing(tx)$          ▷ withdraw coins
5:         if $isValid$ then
6:             $L_{tx} \leftarrow add(L_{tx}, tx)$
7:     else if $tx$ is ready for commitment then
8:         Commit $tx$ and update the ledger
9: if $N_k$ is the leader then
10:    $CL_h \leftarrow < blockTS, L_{tx}, i, h >$
11:    sends $CL_h$ with a quorum certificate to the beacon chain
12: else if $N_k$ is not the leader then
13:    signs $CL_h$ received from the leader
```

Haechi works as a *leader-based* protocol. A potential concern is whether the beacon chain will become the bottleneck and degrade the performance of a sharded system. To evaluate this, we construct related experiments in § VIII, showing that our solution only suffers modest performance losses because of the low-cost ordering operation in the beacon chain and the batching optimization. Furthermore, since the relevant shard chains provide data availability for transactions, the transaction data in the beacon chain can be pruned once they are committed. Haechi will thus not introduce momentous storage overhead into the beacon chain. Overall, since the beacon chain in Haechi neither executes transactions nor maintains the state, it is feasible to expand the beacon chain to carry through the ordering phase. Besides, we also discuss another *leaderless* implementation of Haechi. However, without the coordination of a leader, the leaderless Haechi brings more communication overheads, which will be discussed in § X.

Finalization fairness algorithm. To solve the in-flight transaction challenges in § V-B, we propose a finalization fairness algorithm without any synchronization mechanism. To achieve this, Haechi first uses the *block timestamp* as the ordering indicator that is piggybacked on CrossLinks of shards. The block timestamp indicates the time of starting processing transactions in blocks and thus can be used to decide the processing order for transactions. On the other hand, since the block timestamp is publicly attached to each block via the intra-shard consensus, it prevents the attackers from creating arbitrary timestamps to interfere with the ordering phase. Furthermore, the block timestamp enables Haechi to create a *deterministic global order*, and thus delicately prevents the cyclic order problem that is notorious in previous fairness algorithms [37], [38]. This is because there are no two nodes of the beacon chain receiving different timestamps regarding the same CrossLink. As a result, all nodes can determine a consistent global order based on block timestamps.

Then, we adopt a so-called *at-least-one ordering rule* to guarantee that all in-flight transactions are ordered fairly. The key is to start ordering transactions *only when the beacon chain receives at least one consecutive CrossLink from each shard*. Since the block timestamp is monotonically incremental for each shard chain, a larger block height indicates a larger block timestamp in a shard. Once the beacon chain receives a new CrossLink with the consecutive block height (e.g., the height is $h+1$ where h is the block height of the last received CrossLink) from a shard, it can determine there are no in-flight transactions that are processed earlier than the transactions in the received CrossLink from the shard. When receiving at least one CrossLink from each shard, the beacon chain has a global view of all in-flight transactions. Therefore, the beacon chain definitely learns which transactions can be ordered in the current ordering cycle.

Algorithm 2 illustrates the procedure of our finalization order fairness algorithm. This algorithm runs in each ordering cycle and will create a new beacon chain block containing an m-sized Cross-shard Call Lists $CCLs$. Specifically, $CCL[i]$ corresponds to shard S_i, consisting of a set of ordered OTXs whose called OSCs are maintained by S_i, also as shown in Fig. 5. Note that $CCLs$ is equal to a global order when we merge all OTXs in it. The algorithm consists of three steps:

(a) CrossLinks preparing (Lines 1-11). Every time when the beacon chain BC receives a CrossLink $S_i.CL$ from S_i, it checks if $S_i.CL$ has a consecutive block height with the last CrossLink stored in $shardCLs$ (Line 2). Here $shardCLs$ is an m-sized two-dimension vector maintaining all received CrossLinks whose block heights are consecutive. Specifically, for the corresponding $shardCLs[i]$ of S_i, there are no unreceived CrossLinks of S_i whose block heights are less than the rear of $shardCLs[i]$. This is used to ensure no in-flight CrossLinks skip the current ordering cycle. If $S_i.CL$ doesn't satisfy it, BC will put $S_i.CL$ into the CrossLink pool $CLPool$ (Line 10). Otherwise, BC adds $S_i.CL$ and its followed CrossLinks from $CLPool$ to $shardCLs[i]$ (Lines 4-8). After that, BC updates the latest received timestamp of S_i. The latest received timestamp $shardLastTS[i]$ ensures no in-flight transactions from S_i are processed earlier than currently received transactions from S_i since the block timestamp is incremental with block height (Line 9).

(b) CrossLinks filtering (Lines 12-13). Once receiving at least one consecutive CrossLink from each shard, i.e., $shardCLs$ stores at least one CrossLink for all shards, BC can determine which transactions can be fairly ordered based on the timestamp of their CrossLinks. Specifically, BC chooses those received CrossLinks whose timestamp is no larger than the minimum of the last CrossLinks among shards, i.e., $\min_{1\leq i\leq m}\{shardLastTS[i]\}$. This ensures all transactions that will be ordered in the current ordering cycle correspond with a smaller timestamp than any in-flight transactions. We will give proof in Lemma 1. Intuitively, the *minimum among all shards* indicates the smallest timestamp before which BC can guarantee no in-flight transactions.

(c) Transactions ordering (Lines 14-17). In this step, BC will order all transactions that are chosen in the step of CrossLinks filtering. The ordering rule follows: (i) transactions in a CrossLink with a smaller block timestamp are ordered first; (ii) transactions in the same CrossLink but with smaller indexes in the CrossLink are ordered first. Then, these ordered transactions are appended to the relevant cross-shard call lists $CCLs$ based on what OSC they invoke. $CCLs$ will be used as the content of a new block of the beacon chain and verified via one instance of intra-shard consensus. Finally, each $CCLs[i]$ will be sent to S_i for the next execution phase.

Algorithm 2 The finalization fairness algorithm in Haechi

```
ASSUME: $m$ shard chains
INPUT: a set of CrossLinks, an $m$-sized vector $shardLastTS$, an
    $m$-sized two-dimension vector $shardCLs$
OUTPUT: an $m$-sized $CCLs$ recorded in a new beacon block
    ▶ Step 1: CrossLinks preparing
 1: upon receiving a CrossLink $S_i.CL$ from shard $S_i$ do
 2:     if $S_i.CL.h = shardCLs[i].rear.h + 1$ then
 3:  ▷ $S_i.CL$ has a consecutive block height of the latest added CrossLink
 4:         $shardCLs[i] \leftarrow add\ (shardCLs[i],\ S_i.CL)$
 5: ▷ packs all CrossLinks with block heights that are consecutive with that
    of the last CrossLink in $shardCLs[i]$
 6:         for all $CL \in CLPool$ do
 7:             if $CL.h = shardCLs[i].rear.h + 1$ then
 8:                 $shardCLs[i] \leftarrow add\ (shardCLs[i],\ CL)$
 9:         $shardLastTS[i] \leftarrow shardCLs[i].rear.blockTS$
10:     elseIf $S_i.CL.h > shardCLs[i].rear.h + 1$ then
11:         add $S_i.CL$ to the CrossLink pool $CLPool$
    ▶ Step 2: CrossLinks filtering
12:     if $|shardCLs[i]| \geq 1, \forall 1 \leq i \leq m$ then
13:         Choose all CrossLinks from $shardCLs$ with a block
    timestamp $blockTS \leq \min_{1\leq i\leq m}\{shardLastTS[i]\}$.
    ▶ Step 3: Transactions ordering
14:     for each $CL$ chosen in $Step2$ do
15:         for each $OTX \in CL$ do
16:             if $OTX$ call an OSC maintained by $S_i$ then
17:                 $CCLs[i] \leftarrow addWithOrder(CCLs[i], OTX)$
```

Haechi reduces performance loss caused by the global ordering phase from two aspects. First, our finalization fairness algorithm maintains the transaction throughput by allowing shards to continuously process and commit transactions. As opposed to the strawman solution discussed in § V-C, shards in Haechi are not blocked and impeded by slow-running shards and can commit those transactions irrelevant to OSCs via intra-shard consensus independently. Haechi, therefore, does not sacrifice the transaction throughput. Second, the at-least-one ordering rule reduces the average confirmation time of OTXs by enabling multiple CrossLinks to be ordered in one ordering cycle. Specifically, those CrossLinks generated by the shards with a fast consensus speed may have small timestamps, and the beacon chain will order them in the same ordering cycle if their timestamps satisfy the CrossLinks filtering condition (Lines 12-13). As a result, with more CrossLinks being ordered within an ordering cycle, the average waiting time for each CrossLink to order and commit will decrease.

C. Execution Phase

With deterministic transaction orders received from the beacon chain, a shard can execute the related OSCs with the received cross-shard call list. According to the complexity of an OSC, there are two conditions when executing an OSC:

One-shard execution. If the invoked OSC is a *simple* contract without contract interactions, the shard can execute it independently via an intra-shard consensus. Specifically, consensus nodes use the input data to execute the OSC. Depending on the execution result, the shard labels a tag for each OTX.

(a) Lock to Commit. If the OSC is executed successfully, the shard labels the transaction with a *final-and-commit* tag. Simultaneously, it modifies the local OSC state and locks related data.

(b) Request to Abort. If the shard fails to execute the OSC, the transaction cannot be committed and has to be aborted. In this case, the shard labels the transaction with a *final-but-abort* tag and discards all modifications of the OSC state introduced by the transaction.

Multi-shard execution. If the invoked OSC is a *complex* contract that invokes other contracts in other shards during its execution, the shard handles the OTX via a collaborative execution way. During the process of the collaborative execution, each related shard executes the relevant contract it maintains. As the one-shard execution, the shard labels a tag for the transaction according to the execution result of the OSC that it maintains.

(a) Vote for Commit. If an OSC in a shard is executed successfully, the OTX is labeled with a *progress-and-commit* tag by the shard. Besides, the shard specifies all other OSCs invoked by its OSC (i.e., contract dependency) so that the related shard can determine when to commit the related transaction and contract states. Meanwhile, the shard records the snapshot of the modifications of the OSC state and locks the related data.

(b) Vote for Abort. If, however, a shard fails to execute its OSC, the OTX is labeled with a *progress-but-abort* tag. Similarly, the shard needs to specify the contract dependency for the commitment decision. Besides, the shard discards all modifications of its OSC state.

After handling each OTX in the received cross-shard call list, the shard collects the execution results (commit or abort), return value (if they exist), and contract dependency (for multi-shard execution) and then packs them into an execution message with an attestation. The execution message is next forwarded to the sender shard for the final commitment.

D. Commitment Phase

In the final commitment phase, each transaction sender shard works as the coordinator to help achieve atomic commitment for the transaction. Based on the tags of received execution messages, there are three actions that nodes of the sender shard could take for an OTX:

(a) If receiving a message with a *final-and-commit* tag, meaning that only one contract involves in the handling of the OTX and the contract is executed finally successfully, nodes coordinate atomic commitment by sending related shards a commitment message with a verification. Specifically, the shard that records the OTX commits the transaction and modifies the sender's account permanently, while another shard that maintains the invoked contract will be notified to unlock the related data and commit modifications of the contract state.

(b) If nodes receive a message with a *progress-and-commit* tag, meaning that there are multiple contracts involved in the handling of the OTX, then it needs to wait for receiving all related messages. Specifically, nodes check the contract dependency in the message to determine if they have received all execution results from related shards. Once nodes receive all messages with the *progress-and-commit* tag, they can commit the transaction. The sender shard modifies the sender's account permanently. Meanwhile, all related shards that involve in modifying contracts are informed about unlocking the related data and committing modifications to the contract state.

(c) If nodes receive a message with a *final-but-abort* tag or *progress-but-abort* tag, then the sender shard aborts the transaction and notifies all other related shards to discard the modifications of contracts.

Finally, all OTXs are recorded in the committing list of shard blocks after the commitment phase.

VII. Security Analysis

We next prove the finalization fairness, safety, and liveness of Haechi.

A sharded system is a multi-chain ledger where each shard maintains a blockchain via an intra-shard BFT consensus protocol. Each shard can therefore be considered a BFT State Machine Replication (BFT-SMR). However, since cross-shard transactions can modify the data of more than one shard simultaneously, it is not adequate to use the original safety and liveness property (cf. intra-shard safety and liveness in § IV-B) of BFT-SMR to completely describe the security of a sharded system. Therefore, we first define safety and liveness for a sharded system.

Definition 4 (Sharded System Safety)**.** *A sharded system is said to satisfy the safety property if: (i) any two honest nodes of the same shard maintain the same prefix ledger; (ii) any two honest nodes from two different shards have the same commitment sequence and same operation (commit or abort) for all cross-shard transactions involving the two shards.*

Definition 5 (Sharded System Liveness)**.** *Every transaction received by at least one honest node will be eventually handled by relevant shards and get a response from the sharded system.*

A. Finalization Fairness Analysis

Recall from the definition in § IV-C, finalization fairness indicates that if two transactions are involved in calling the same contract, then their execution orders (i.e., finalization orders) in all contracts they are jointly involved in are the same as their processing order. In Haechi, transaction execution and commitment follow the global order established in the ordering phase. Therefore, we can prove that Haechi provides finalization fairness by showing that the ordering of transactions in the ordering phase follows the processing order strictly. Note that the beacon chain works in ordering cycles, in each of which the beacon chain picks received CrossLinks selectively and orders OTXs in them.

Lemma 1. *In Haechi, the block timestamp of any in-flight CrossLink is larger than all block timestamps of CrossLinks that are ordered in the current ordering cycling.*

Proof: We prove this lemma by comparing: (i) in-flight CrossLinks with the received CrossLinks from the same shard; (ii) in-flight CrossLinks with the received CrossLinks from different shards.

(i) First, in our finalization fairness algorithm (Algorithm 2), the beacon chain starts ordering in the current ordering cycle once it receives at least one CrossLink with a consecutive block height from each shard. On the one hand, *at least one* ensures that the beacon chain will not miss CrossLinks from any shard in the current ordering cycle. On the other hand, the *consecutive block height* ensures that all CrossLinks corresponding to blocks with smaller block heights have been received in the current ordering cycle or ordered in the previous ordering cycles. Therefore, for every shard, all its in-flight CrossLinks have a higher block timestamp than its CrossLinks received by the beacon chain in the current ordering cycle.

(ii) Second, assume $S_i.CL_{last}$ is the last received CrossLink and $\{S_i.CL_{flight}\}$ is a set of all in-flight CrossLinks from S_i. In the current ordering cycle, the largest timestamp that is allowed to be ordered is the minimum block timestamp of the last received CrossLinks among all shards. Without loss of generality, we assume $S_1.CL_{last}$ has the minimum block timestamp in the current ordering cycle, and there are m shard chains. Then, we have:

$$S_1.CL_{last}.blockTS \leq S_i.CL_{last}.blockTS, \qquad (1)$$

where $1 < i \leq m$. Moreover, according to Proof (i) we can know all in-flight CrossLinks of a shard have a higher block timestamp than the shard's last CrossLink that is received by the beacon chain. Thus, for all shards S_i ($1 \leq i \leq m$) and any CrossLink $cl \in \{S_i.CL_{flight}\}$, we have:

$$S_i.CL_{last}.blockTS < cl.blockTS. \qquad (2)$$

Finally, from Equations (1) and (2), we can finally show:

$$S_1.CL_{last}.blockTS \leq cl.blockTS, \qquad (3)$$

where $\forall cl \in \{S_i.CL_{flight}\}$ and $1 \leq i \leq m$. In other words, the block timestamp of any in-flight CrossLink is larger than all block timestamps of CrossLinks that are ordered in the current ordering cycling. □

Lemma 2. *For any two received transactions OTX^1 and OTX^2, piggybacked by CrossLinks CL_1 and CL_2 respectively, if (i) $CL_1.blockTS < CL_2.blockTS$, or (ii) $CL_1 = CL_2$ and $f_{idx}(OTX^1) < f_{idx}(OTX^2)$, then Haechi must order OTX^1 before OTX^2 in the ordering phase.*

Proof: Note that $f_{idx}(\cdot)$ in Lemma 2 is a function returning the indexes of transactions in a block, same as that of Definition 1. Moreover, conditions (i) and (ii) in Lemma 2 are actually consistent with the ordering rules we defined in the finalization ordering algorithm (cf. § VI-B step (c)). We now prove Lemma 2 by analyzing all possible byzantine behaviors performed by a malicious leader in the beacon chain.

On the one hand, since the ordering phase is driven by an instance of intra-shard consensus, and $f_{idx}(\cdot)$ and block timestamps are publicly verifiable, a new block created by a malicious leader will not be accepted by other consensus nodes if it contains a transaction order violating the ordering rules, e.g., a transaction with a larger index is ordered before another transaction in the same CrossLink. On the other hand, the malicious leader may deliberately drop some CrossLinks it received, leading to transactions in these dropped CrossLinks are not ordered in the appropriate ordering cycles, thus violating the ordering rules. However, consensus nodes can detect such an omission behavior by checking if the selected CrossLinks in the new block have consecutive block heights for each shard and if the last CrossLinks of all shards are considered in the new block. Overall, any byzantine behaviors violating the ordering rules can be detected and prevented. Since the ordering rules are satisfied all the time, Lemma 2 is correct and the proof is done. □

With Lemmas 1 and 2, we now prove Theorem 3:

Theorem 3. *Haechi provides the finalization fairness for a sharded system.*

Proof: Lemmas 1 and 2 guarantee all in-flight and received CrossLinks must be ordered based on the ordering rules. Since the block timestamp and transaction indexes in the ordering rules indicate the processing order, the beacon chain will keep this order for the following execution and commit phases. Therefore, Haechi ensures that the processing order is consistent with the execution order, i.e., ensuring the finalization fairness. □

B. Safety Analysis

We show that Haechi guarantees the sharded system safety property by respectively showing Haechi satisfies the two conditions in Definition 4.

Lemma 4. *For every two honest nodes N_i^1 and N_i^2 with local ledgers $\mathfrak{L}_\mathrm{i}{}^1$ and $\mathfrak{L}_\mathrm{i}{}^2$ in any shard S_i, Haechi guarantees: for all block height h ($0 \leq h \leq min\{f_{len}(\mathfrak{L}_\mathrm{i}{}^1), f_{len}(\mathfrak{L}_\mathrm{i}{}^2)\}$), $\mathfrak{L}_\mathrm{i}{}^1[h] = \mathfrak{L}_\mathrm{i}{}^2[h]$, where $f_{len}(\cdot)$ computes the current block height of a chain.*

Proof: Since each shard runs a BFT consensus for maintaining its own ledger and each shard is controlled by the honest nodes (see the trust assumptions in § IV-B), the shard satisfies the intra-shard safety according to the safety properties of BFT-based protocol. Furthermore, Haechi does not violate the security of the intra-shard consensus. Therefore, N_i^1 and N_i^2, which participate in the same shard, have a consistent ledger in Haechi. □

Lemma 5. *If an OTX involves the modification of several data managed by multiple shards, then all the relevant shards either commit their local modification or abort it.*

Proof: Recall from § VI that the transaction sender's shard in Haechi is responsible for coordinating the cross-shard consensus of an OTX. Once nodes of the sender's shard receive messages from all shards relevant to the transaction, the sender's shard runs a BFT-based intra-shard consensus to make a decision based on the received messages. Since the sender's shard is controlled by the honest nodes, such a consensus can be completed securely and generate one commitment decision for the transaction. Since there is no equivocation, all relevant shards will receive a consistent commitment decision on whether to commit or abort their local modification for the transaction. Moreover, the finalization fairness provided by Haechi (cf. Theorem 3) ensures the transaction is eventually committed or aborted. □

With Lemmas 4 and 5, we now prove Theorem 6:

Theorem 6. *Haechi guarantees the sharded system safety.*

Proof: If the two nodes belong to the same shard, by Lemma 4, they can keep a consistent ledger via the intra-shard consensus protocol. In other words, these two nodes have the same commitment sequence and operation (i.e., commit or abort) for all transactions recorded in their shard chain. On the other hand, if the two nodes belong to different shards, by Lemma 5, they can take consistent action to every OSC transaction via Haechi, i.e., either all commit or all abort transactions. Furthermore, since Haechi ensures finalization fairness where the processing order is the same as the commitment order, these two nodes will have the same commitment sequence for all OSC transactions they involve. Together, these observations imply the desired result, i.e., Haechi satisfies the sharded system safety property. □

C. Liveness Analysis

We now give brief proof to show that Haechi satisfies the liveness property of a sharded system.

Theorem 7. *Haechi guarantees the sharded system's liveness.*

Proof: Based on the type of transaction, there are two conditions for handling a transaction TX:

(i) TX is a non-OTX. In this case, TX will be handled via the original consensus protocol of the sharded system. Specifically, if TX is an intra-shard transaction of a shard, then the shard is responsible for handling it via the intra-shard consensus protocol. If TX is a cross-shard transaction, then all relevant shards collaboratively handle it via the 2P cross-shard consensus protocol. But in either case, the user can receive a response from the sharded system, which is ensured by the underlying protocol of the sharded system.

(ii) TX is an OTX. As messages between two honest parties cannot be dropped, the beacon chain eventually receives a CrossLink containing TX. Besides, the finalization fairness provided by Haechi guarantees that there is no CrossLink being dropped (otherwise, it violates the finalization fairness since transactions in the dropped CrossLinks cannot be ordered fairly). Since an honest shard is responsive according to the intra-shard liveness provided by the intra-shard BFT protocol. Haechi ensures that any OTX is handled and can get a response eventually.

According to the above analysis, we can conclude that Haechi satisfies the sharded system liveness property. □

VIII. Evaluation

In this section, we will first present implementation details and experimental settings. Then, we evaluate and compare Haechi with other cross-shard protocols in terms of the ability to prevent front-running attacks, practicality, and performance.

Implementation. We implement Haechi based on Tendermint [39]. Tendermint provides modular designs for building blockchains, which allows us to smoothly realize our fair ordering algorithm and cross-shard communication without compromising the security of its consensus protocol. In our implementation, each shard runs the Tendermint BFT protocol [14] to reach a consensus on new blocks, and each node of the same shard maintains the same blockchain ledger that stores transaction data in LevelDB. For cross-shard communications, we assign a leader node of each shard as the cross-shard portal, responsible for sending/receiving requests from other shards to avoid redundancy message delivery. New blocks in a shard chain are created by the shard proposer randomly, while the blocks generation in the beacon chain incorporates the finalization fairness algorithm (Algorithm 2) to achieve fair ordering for OSC transactions across shards. We implement multiple kinds of Clients to securely send transactions via the interface provided by Tendermint, where all transactions will be verified before processing.

Baselines for comparison. For comparison, we implement three cross-shard protocols based on Tendermint: Haechi-sync, AHL [17], and Byshard [18]. Haechi-sync is the strawman solution we discussed in § V-C, where shards are blocked and cannot process new transactions until the beacon chain finishes the ordering phase of the last round (i.e., all shards synchronize to the same block height). AHL and Byshard are two state-of-the-art 2P cross-shard protocols vulnerable to front-running attacks. More importantly, they support smart contract-type transactions that are considered in our scenario. Both of them delegate a shard as the cross-shard coordinator. The main difference between them is that AHL uses a dedicated *reference shard* to coordinate the processing of all cross-shard transactions, while Byshard asks the sender shard to be the cross-shard coordinator. The total implementation involves roughly 4.7K lines of code in Golang. In this section, we will evaluate the presented front-running attack and the performance of these cross-shard protocols.

Setup. We run our evaluation in a geo-distributed environment spread across 10 AWS regions in the world (4 in America, 3 in Asia, and 3 in Europe), which includes ten c5a.8xlarge EC2 instances and each one has 32 CPUs, 64 GB RAM, and a 10 Gbps network connection. The sender address and receiver's address of a transaction is generated randomly to guarantee each shard has a balancing workload. The number of nodes in a shard (including the beacon shard) is set to 30, and shards have been running for over 3 minutes for each metrics testing. The mempool of each node can store up to 50,000 transactions, with a maximum byte size of 1 GB. We deploy multiple clients for sending transactions.

Metrics. We use the following metrics for evaluation: (i) attack success rate: the effectiveness of the cross-shard front-running attack; (ii) transaction throughput: the number of transactions a system can execute per second, i.e., TPS in short; (iii) intra-shard latency: the confirmation latency of intra-shard transactions, starting from the time when the client sends an intra-shard transaction to the time when the transaction is committed by a shard; (iv) cross-shard latency: the confirmation latency of cross-shard transactions, starting from the time when the client sends a cross-shard transaction to the time when the transaction is committed by all relevant shards; (v) CCLs interval: the generation interval of cross-shard call lists.

A. Evaluation of Front-running Attack

We first evaluate the effectiveness of our presented cross-shard front-running attack. To achieve this, we implement the front-running attacking behaviors as illustrated in Fig. 4 on

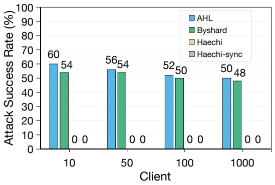


Fig. 7. Attack success rate under varying workloads.

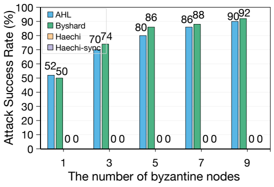


Fig. 8. Attack success rate under varying byzantine ratios.

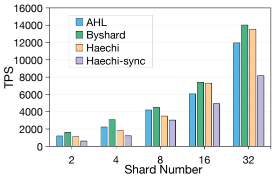


Fig. 9. Throughput under various numbers of shards.

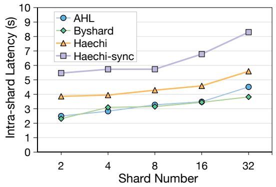


Fig. 10. Confirmation latency of intra-shard transactions.

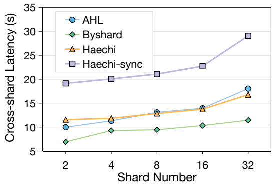


Fig. 11. Confirmation latency of cross-shard transactions.

all evaluated protocols. Specifically, we implement byzantine nodes that create front-running intra-shard transactions once they observe victims' cross-shard transactions are processed during intra-shard consensus. We then trace the execution order between the front-running transactions and the victims' transactions from log files. We evaluate the effectiveness of the front-running attack with the attack success rate, which represents a ratio of the front-running transactions that are executed before the victims' transactions.

Fig. 7 and 8 show the attack success rate of four compared protocols under different settings, where we create 50 victims' transactions and repeat 5 times for each experimental setting. In Fig. 7, we evaluate the impact of transaction workloads on the attack success rate. We initiate varying numbers of clients to send transactions to the contract shard. For 2P cross-shard protocol, Fig. 7 demonstrates that with higher workloads on the contract shard, the attack success rate decreases. This is because once a contract shard suffers from a heavy workload, front-running transactions cannot be processed immediately and may thus be processed with victims' cross-shard transactions in the same instance of intra-shard consensus. Fig. 8 shows the impact of the ratio of byzantine nodes on the attack success rate, where we initiate 1, 3, 5, 7, and 9 byzantine nodes (out of 30 nodes) in each shard. The experiments show that if more front-running attackers exist in a shard, the possibility of being front-run will be higher in 2P cross-shard protocols. In contrast, both Haechi and Haechi-sync can prevent such attacks effectively and we did not observe any successful front-running attacks under a realistic test environment.

B. Scalability

We then evaluate the scalability of different cross-shard protocols, where we compute transaction throughput, intra-shard and cross-shard latency under the number of shards $m = 2, 4, 8, 16, 32$. To measure TPS, we count the number of valid transactions that are packed and handled in a block via the log files generated by the ledger. To measure confirmation latency, we trace some transactions and compute the delay from when it was sent to when it was committed.

As shown in Fig. 9, 10, 11, Haechi scales well as the other two 2P cross-shard protocols. The TPS of Haechi increases nearly linearly while the intra-shard and cross-shard confirmation latency increase slowly with the increasing number of shards. Haechi can achieve 13,000+ TPS, 6s intra-shard latency, and 17s cross-shard latency with 32 shards. In all cross-shard protocols, the performance of Byshard is the best regardless of the number of shards since it amortizes the overhead of cross-shard communications to every shard. Furthermore, AHL performs worse than Haechi with increasing shard numbers, as it requires one reference shard to receive and coordinate all cross-shard transactions, which will lead to a bottleneck of the reference shard. Haechi-sync also does not scale well because of its synchronization mechanism. In contrast, the ordering phase performed by Haechi does not introduce significant overhead because of the finalization fairness algorithm and low-cost ordering operations.

TABLE I. Evaluation of extra gas consumption of a Haechi-enabled sharded system with 32 shard chains.

Protocol	#CrossLink	MAX	AVG	MIN
Haechi-sync	32	250 gas	39 gas	8 gas
Haechi	10,000	78,000 gas	103 gas	8 gas

C. Micro-benchmark

To evaluate the practicality of the global ordering phase introduced by both Haechi and Haechi-sync, we construct two experiments to calculate the CCLs interval and gas consumption respectively. The CCLs interval reflects the time spent in one ordering cycle, and the gas consumption measures how much extra gas will be used by the ordering phase.

First, we run both Haechi and Haechi-sync to wait for them to generate at least 10 CCLs, and compute the maximum, minimum, and average intervals for two consecutive CCLs. Fig. 12 shows that the impact of the number of shards on CCLs generations. Due to the finalization fairness algorithm used in Haechi, the number of shards brings a subtle impact on CCLs generations, and the CCLs interval is acceptable under a real-world network latency, e.g., only 1.68s on average with 32 shards. In contrast, Haechi-sync introduces higher overhead into its ordering phase due to its synchronization mechanism.

Then, to evaluate the practicality of our protocol on gas consumption, we convert the ordering phase into the number of comparisons of timestamps. Specifically, the ordering phase uses a sorting algorithm to order CrossLinks based on their timestamps, which involves a series of comparison operations. Therefore, we implement and deploy a comparison contract to estimate the gas consumption of one comparison for two timestamps, showing that one comparison operation only costs 780 gas. Besides, we assume the average number of transactions per CrossLink is 100 (note that transactions will amortize the gas), and at most 10,000 CrossLinks[8] are ordered in one ordering cycle of Haechi. Note that Haechi-sync (with 32 shards) always orders 32 CrossLinks in each ordering cycle. With these data, we can estimate the extra gas consumption needed in the ordering phase for each transaction.

[8] 10,000 means when receiving the first CrossLink from the slowest shard, the beacon chain has already received 9,999 CrossLinks from other shards, indicating > 300 times difference in consensus speeds between shards.

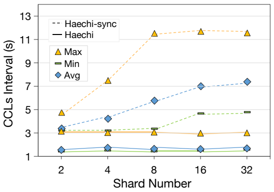


Fig. 12. The generation interval of CCLs.

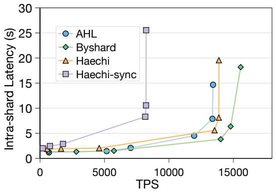


Fig. 13. TPS vs. intra-shard latency.

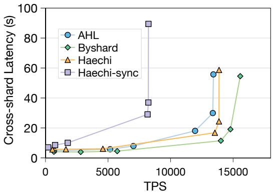


Fig. 14. TPS vs. cross-shard latency.

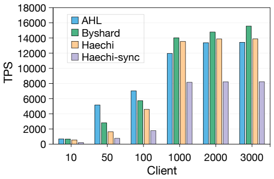


Fig. 15. TPS under varying numbers of clients.

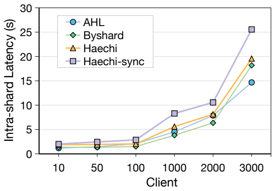


Fig. 16. Intra-shard latency under varying clients.

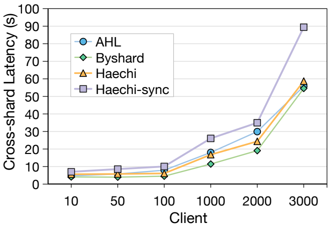


Fig. 17. Cross-shard latency under varying clients.

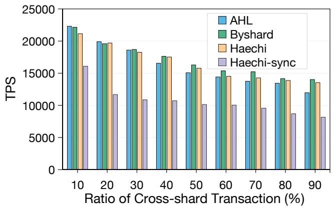


Fig. 18. TPS under varying cross-shard ratios.

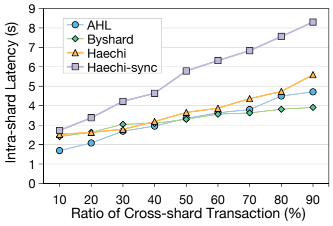


Fig. 19. Intra-shard latency under varying ratios.

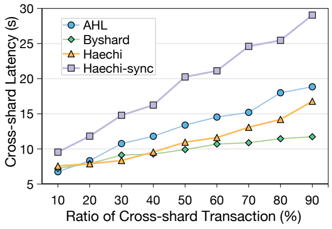


Fig. 20. Cross-shard latency under varying ratios.

Table I shows the upper bound of gas consumption, where MAX, AVG, and MIN are respectively corresponding to the complexities of a sorting algorithm in different cases, i.e., $O(n^2)$, $O(nlogn)$, and $O(n)$ times of comparisons with n CrossLinks. From the historical gas price of Ethereum from July 2022 to July 2023[9], the maximum gas price is about 150 Gwei[10]. Therefore, the upper bound of gas consumption for each transaction in Haechi is worth $78,000 * 150$ Gwei $=$ $1.17*10^{-2}$ ETH. Similarly, we can calculate the average fee in Haechi, $1.55 * 10^{-5}$ ETH, which is very cheap and negligible to the asset loss of users from a front-running attack.

D. Performance

We next evaluate the performance of Haechi under different experimental settings. In these experiments, we deploy 33 shards (including 1 beacon shard) and use clients to continuously send transactions to derive the peak performance.

Fig. 13 and 14 show the relationship between TPS and confirmation latency. We can find that Haechi only compromises slight performance decreasing compared to Byshard, and even performs better than AHL; while Haechi-sync sacrifices marked performance loss. Haechi has less than 5s confirmation latency for intra-shard transactions and less than 10s confirmation latency for cross-shard transactions when the system has a TPS of less than 15,000, which can satisfy most requirements of decentralized applications in a practical scenario.

Fig. 15, 16, and 17 present the performance of different cross-shard protocols under distinct workloads, in which we respectively instantiate $c = 10, 50, 100, 1000, 2000, 3000$ clients for a shard to establish connections with the shard nodes and send transactions to them. These experiments are used to simulate a real-world scenario where users concurrently send requests to the system. The results show that in our experimental environment, all cross-shard protocols achieve the peak TPS when the number of clients in a shard exceeds 2,000. But the confirmation latency will still increase with more clients. This is because transactions need to wait more time in the mempool once the speed of transaction requests is larger than the speed of handling transactions by the system. Besides, we observe committing a new block ranges from 1s to 8s in different shards because of region delays and transaction workloads. This shows the existence of the consensus difference in a realistic sharded system.

Fig. 18, 19, and 20 give the impact of the ratio of cross-shard transactions on the performance, where we set this ratio from $r = 10\%$ to 90%. For example, $r = 90\%$ represents that an average of 9 out of 10 transactions are cross-shard transactions. From the experimental results, we can find that with an increasing ratio of cross-shard transactions, the performance of a sharded system will decrease. This is because handling cross-shard transactions will introduce many overheads in communication and computation, which reflects the significance of designing an efficient cross-shard protocol. We hope our designed asynchronous and non-blocking processing for cross-shard transactions (§ VI-A) can motivate readers to explore more efficient cross-shard protocols.

From the above evaluations, we can find Haechi only sacrifices a slight performance decrease compared to the 2P cross-shard consensus protocol while Haechi also provides finalization fairness property for a sharded system.

IX. Related Work

Blockchain sharding. Sharding has been proposed to enhance blockchain scalability. A major new challenge is how to handle cross-shard transactions. Many works [4], [33] adopt the optimistic commit protocol to guarantee isolation and atomicity, where cross-shard transactions are divided into several sub-transactions to related shards and committed optimistically. However, this protocol needs to abort and roll back all transactions if there are conflicting read/write of the same data. Another cross-shard protocol is the two-phase (2P) protocol as we discussed above, which is adopted by most works [2], [3], [17]–[19], [31], [40].

[9] https://etherscan.io/chart/gasprice

[10] 1 Gwei=10^{-9} ETH

Due to the ledger separation, handling a cross-shard transaction inevitably involves several steps. In each step, a related shard runs one instance of intra-shard consensus to handle related data. Thus, compared to intra-shard transactions, handling cross-shard transactions with the previous cross-shard consensus protocol introduces several instances of the intra-shard consensus. To the best of our knowledge, none of the previous works care about the differences between intra-shard and cross-shard consensus protocols. However, as we discussed above, these differences will introduce the finalization fairness problem to a sharded system and bring adverse effects that will further limit the application of sharding technology. Instead, Haechi can address the finalization fairness problem without compromising too much performance.

Order-fairness Protocols. With the popularity of DeFi, a new property called *order fairness* is gaining more and more attention. Many recent works have shown that adversaries can make profits by manipulating transaction order [21], [22], [24], [25], [41], [42]. Accordingly, various fair consensus protocols have been proposed to achieve order fairness for blockchains. Some of them [37], [38], [43]–[45] adopt the *time-based* order-fairness, where transactions are ordered based on the time when they are received by nodes. The others [46]–[48] adopt the *blind order-fairness*, where the content of transactions is hidden until they are committed to a total ordering, and thus transaction order is random.

All the above works consider how to form order fairness during the period from when transactions appear in the mempool to when transactions are packed into blocks. However, Haechi aims to achieve finalization fairness in a sharded system, focusing on order fairness during the period from when transactions are first processed in the sender shards to when transactions are executed in the contract shards. Besides, most of them assume a permissioned blockchain scenario, which only allows a limited number of consensus nodes. In contrast, Haechi provides a new solution for achieving order fairness in permissionless blockchains.

X. Discussion

Block timestamp security. In § VI-B, we rely on the correct block timestamp to achieve finalization fairness. While all prominent blockchain designs ensure that block proposers cannot set arbitrary timestamps [49], [50], the proposers still have small leverage to modify the timestamp within tolerance introduced by partial synchrony or asynchrony so that they can influence the execution order. Nevertheless, such a minor-level timestamp manipulation seems unavoidable across fairness protocols that use timestamps as the ordering indicator even with synchronized clocks [37], [38], [45]. A mechanism to mitigate this risk is to hide transaction information using an encryption scheme until its block reaches a consensus. We denote T_c as the time used for this consensus process. In this case, when the front-running proposer observes the content of the transaction, it needs to attach a timestamp with a time deviation of larger than T_c compared to the actual time. However, such a marked deviation of a block timestamp could lead to rejection on this block by honest nodes.

Beacon chain-free Haechi. Currently, Haechi assigns the beacon chain to coordinate/lead the ordering phase and establish a globally fair order via our finalization fairness algorithm. However, we emphasize our algorithm can be extended to any sharded system even if it has not the beacon chain. This will formulate a so-called *beacon chain-free Haechi* or *leaderless Haechi*. Specifically, instead of sending CrossLinks to the specific beacon chain, all shards need to broadcast their CrossLinks to other shards. Then each shard can run the finalization fairness algorithm locally to establish a global order. Note that our algorithm establishes a *deterministic* global order as we use the block timestamps as the ordering indicator and the block timestamps are globally consistent, i.e., for any node, the timestamp of a block is consistent. With the deterministic guarantee, all shards can eventually achieve a consistent view of the global order. However, this all-to-all communication brings significantly more communication overhead as compared to the current (all-to-one and then one-to-all) beacon/leader chain-based approach.

Intra-shard order fairness. This work focuses on the finalization fairness of sharded systems, i.e., the execution order of transactions is the same as their processing order. More precisely, we guarantee *cross-shard order fairness* against the front-running attack presented in this paper. However, the adversary can manipulate the transaction order in one shard chain, breaking *intra-shard order fairness*. Specifically, the adversary controls the intra-shard consensus of the shard to order their transactions before the victim's transactions in an adversary shard block. In this case, the adversary manipulates the processing order, thus affecting the finalization fairness.

Nevertheless, the order fairness problem for intra-shard consensus can be mitigated by existing order-fairness protocols as discussed in § IX. Haechi focuses on finalization fairness across shards, we omit the intra-shard order fairness from our evaluations. However, we emphasize that these order-fairness protocols are complementary and compatible with Haechi and can be easily integrated into Haechi.

Multi-shard transaction order. In a real-world scenario, a transaction can involve multiple smart contracts by *contract interactions*, where some contract operations, e.g., function-call and data-return, will invoke other contracts. Such a transaction is also called *multi-shard contract transaction* in a sharded system if its involved contracts are managed by multiple shards. A multi-shard transaction will be handled by more than two shards and may involve at least two block timestamps until being executed in the OSC. Haechi can still ensure finalization fairness by using the *first block timestamp* when the multi-shard transaction is first packed into a shard block. In the implementation, the beacon chain maintains a mapping data structure to trace the first block timestamp for multi-shard transactions so that it can ensure the execution order is consistent with the processing order (i.e., ensure finalization fairness). There are some potential optimizations for it; however, they are beyond the scope of this paper.

Impact of reconfiguration on the attack. The proposed front-running attack relies on the attacker observing the victim's transactions in time when they are in the same shard. It is part of the natural consensus flow of communication across shards. While the attack is in process, if the attacker is swapped to another shard via reconfiguration, it may not be able to successfully complete the attack. However, if nodes are reorganized so frequently, then a system may not be able to process transactions either, which is impractical. For instance, the reconfiguration interval (called *epoch*) for Ethereum is about 384 seconds as each epoch contains 32 blocks. Therefore, attackers still have the advantage of launching front-running attacks in their shards. Reconfiguration cannot compromise the effectiveness of the front-running attack without reducing the system throughput to near zero.

XI. Conclusion

In this paper, we explore the finalization fairness problem in sharded systems. We present, to the best of our knowledge, the first front-running attack targeting the existing cross-shard consensus protocols. Then, we propose Haechi, a new cross-shard consensus protocol that provides finalization fairness for users from different shards. Haechi introduces a global ordering phase to prevent attackers from intentionally manipulating the execution order of transactions calling those smart contracts that are vulnerable to front-running attacks in sharded systems. We implement a full prototype and compare Haechi to other cross-shard consensus protocols. Our results show that Haechi can achieve approximate throughput and confirmation latency as other cross-shard protocols (i.e., 13,000+ TPS and 17s cross-shard latency with 33 shards). We also evaluate the presented cross-shard front-running attacks. Experimental results show the effectiveness of the attacks on existing cross-shard protocols while Haechi can prevent such attacks well.

Acknowledgement

The authors would like to thank all the anonymous reviewers for their valuable comments to improve this paper. The work is partially supported by the National Key Research and Development Plan (2021YFB2700302), the National Natural Science Foundation of China (62172453), the National Natural Science Foundation of Guangdong province (2022A1515010154), the Major Key Project of PCL (PCL2021A06), the Pearl River Talent Recruitment Program (No. 2019QN01X130), and the National Science Foundation (NSF) under grant CNS1846316.

References

[1] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, “A secure sharding protocol for open blockchains,” in *ACM CCS*, 2016, pp. 17–30.

[2] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “Omniledger: A secure, scale-out, decentralized ledger via sharding,” in *IEEE S&P*, 2018, pp. 583–598.

[3] M. Zamani, M. Movahedi, and M. Raykova, “Rapidchain: Scaling blockchain via full sharding,” in *ACM CCS*, 2018, pp. 931–948.

[4] J. Wang and H. Wang, “Monoxide: Scale out blockchains with asynchronous consensus zones,” in *NSDI*, 2019, pp. 95–112.

[5] B. David, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, “Gearbox: Optimal-size shard committees by leveraging the safety-liveness dichotomy,” in *ACM CCS*, 2022, pp. 683–696.

[6] R. Rana, S. Kannan, D. Tse, and P. Viswanath, “Free2shard: Adversary-resistant distributed resource allocation for blockchains,” *ACM POMACS*, vol. 6, no. 1, pp. 1–38, 2022.

[7] M. Król, O. Ascigil, S. Rene, A. Sonnino, M. Al-Bassam, and E. Rivière, “Shard scheduler: object placement and migration in sharded account-based blockchains,” in *ACM AFT*, 2021, pp. 43–56.

[8] Ethereum, “Ethereum sharding,” https://ethereum.org/en/eth2/shard-chains/, accessed: 2023.

[9] Harmony, “Harmony: Technical whitepaper,” https://harmony.one/pdf/whitepaper.pdf, 2018.

[10] Zilliqa, “The zilliqa technical whitepaper,” https://docs.zilliqa.com/whitepaper.pdf, 2017.

[11] Elrond, “Elrond: A highly scalable public blockchain via adaptive state sharding and secure proof of stake,” https://elrond.com/assets/files/elrond-whitepaper.pdf, 2019.

[12] A. Skidanov and I. Polosukhin, “Nightshade: Near protocol sharding design,” https://nearprotocol.com/downloads/Nightshade.pdf, 2019.

[13] Ontology, “Ontology sharding, draft v0.2,” https://github.com/ontio/documentation/blob/master/sharding/ontology-sharding.pdf, 2019.

[14] E. Buchman, “Tendermint: Byzantine fault tolerance in the age of blockchains,” Ph.D. dissertation, University of Guelph, 2016.

[15] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hotstuff: Bft consensus with linearity and responsiveness,” in *ACM PODC*, 2019, pp. 347–356.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[17] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, “Towards scaling blockchain systems via sharding,” in *ACM SIGMOD*, 2019, pp. 123–140.

[18] J. Hellings and M. Sadoghi, “Byshard: sharding in a byzantine environment,” *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2230–2243, 2021.

[19] M. J. Amiri, D. Agrawal, and A. El Abbadi, “Sharper: Sharding permissioned blockchains over network clusters,” in *ACM SIGMOD*, 2021, pp. 76–88.

[20] G. Pîrlea, A. Kumar, and I. Sergey, “Practical smart contract sharding with ownership and commutativity analysis,” in *ACM PLDI*, 2021, pp. 1327–1341.

[21] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, “High-frequency trading on decentralized on-chain exchanges,” in *IEEE S&P*, 2021, pp. 428–445.

[22] S. Eskandari, S. Moosavi, and J. Clark, “Sok: Transparent dishonesty: front-running attacks on blockchain,” in *FC Workshops*. Springer, 2019, pp. 170–189.

[23] C. F. Torres, R. Camino *et al.*, “Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain,” in *USENIX Security*, 2021, pp. 1343–1359.

[24] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, “Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability,” in *IEEE S&P*, 2020, pp. 910–927.

[25] K. Qin, L. Zhou, and A. Gervais, “Quantifying blockchain extractable value: How dark is the forest?” in *IEEE S&P*, 2022, pp. 198–214.

[26] Uniswap.io, “uniswap,” https://docs.uniswap.io/, accessed: 2023.

[27] Solidity, “Solidity language,” http://solidity.readthedocs.io, 2018.

[28] Y. Chen, Z. Guo, R. Li, S. Chen, L. Zhou, Y. Zhou, and X. Zhang, “Forerunner: Constraint-based speculative transaction execution for ethereum,” in *ACM SOSP*, 2021, pp. 570–587.

[29] Z. Hong, S. Guo, E. Zhou, J. Zhang, W. Chen, J. Liang, J. Zhang, and A. Zomaya, “Prophet: Conflict-free sharding blockchain via byzantine-tolerant deterministic ordering,” in *IEEE INFOCOM*, 2023.

[30] E. Androulaki, C. Cachin, A. D. Caro, and E. Kokoris-Kogias, “Channels: Horizontal scaling and confidentiality on permissioned blockchains,” in *ESORICS*. Springer, 2018, pp. 111–131.

[31] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, “Chainspace: A sharded smart contracts platform,” in *NDSS*, 2018.

[32] M. Castro, B. Liskov *et al.*, “Practical byzantine fault tolerance,” in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[33] S. Das, V. Krishnan, and L. Ren, “Efficient cross-shard transaction execution in sharded blockchains,” *arXiv preprint arXiv:2007.14521*, 2020.

[34] Ethereum, “The beacon chain,” https://ethereum.org/en/upgrades/beacon-chain/, accessed: 2023.

[35] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *EuroSys*, 2022, pp. 34–50.

[36] K. Wüst, S. Matetic, S. Egli, K. Kostiainen, and S. Capkun, "Ace: Asynchronous and concurrent execution of complex smart contracts," in *ACM CCS*, 2020, pp. 587–600.

[37] Y. Zhang, S. Setty, Q. Chen, L. Zhou, and L. Alvisi, "Byzantine ordered consensus without byzantine oligarchy," in *OSDI*, 2020, pp. 633–649.

[38] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *CRYPTO*. Springer, 2020, pp. 451–480.

[39] D. Cason, E. Fynn, N. Milosevic, Z. Milosevic, E. Buchman, and F. Pedone, "The design, architecture and performance of the tendermint blockchain network," in *IEEE SRDS*, 2021, pp. 23–33.

[40] J. Zhang, Z. Hong, X. Qiu, Y. Zhan, S. Guo, and W. Chen, "Skychain: A deep reinforcement learning-empowered dynamic blockchain sharding system," in *ACM ICPP*, 2020, pp. 1–11.

[41] L. Zhou, K. Qin, A. Cully, B. Livshits, and A. Gervais, "On the just-in-time discovery of profit-generating transactions in defi protocols," in *IEEE S&P*, 2021, pp. 919–936.

[42] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) attacks," *IEEE S&P*, 2023.

[43] K. Kursawe, "Wendy, the good little fairness widget: Achieving order fairness for blockchains," in *ACM AFT*, 2020, pp. 25–36.

[44] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, strong order-fairness in byzantine consensus," *Cryptology ePrint Archive*, 2021.

[45] C. Cachin, J. Mićić, N. Steinhauer, and L. Zanolini, "Quick order fairness," in *FC*. Springer, 2022, pp. 316–333.

[46] A. Asayag, G. Cohen, I. Grayevsky, M. Leshkowitz, O. Rottenstreich, R. Tamari, and D. Yakira, "A fair consensus protocol for transaction ordering," in *IEEE ICNP*, 2018, pp. 55–65.

[47] C. Stathakopoulou, S. Rüsch, M. Brandenburger, and M. Vukolić, "Adding fairness to order: Preventing front-running attacks in bft protocols using tees," in *IEEE SRDS*, 2021, pp. 34–45.

[48] D. Malkhi and P. Szalachowski, "Maximal extractable value (mev) protection on a dag," *arXiv preprint arXiv:2208.00940*, 2022.

[49] Aptos, "The aptos blockchain," https://github.com/aptos-labs/aptos-core/blob/main/developer-docs-site/static/papers/whitepaper.pdf, accessed: 2023.

[50] Ethereum, "Ethereum timestamp," https://ethereum.org/en/developers/docs/data-and-analytics/block-explorers/, accessed: 2023.

Appendix A
A more flexible Front-running Attacking Model in Sharded Systems

In section III-A, we give a concrete example that the adversary launches a front-running attack by utilizing the processing-execution difference between intra-shard and cross-shard transactions. Briefly, the adversary creates intra-shard transactions to directly call a contract via intra-shard consensus, once it monitors that the victims' cross-shard transactions from other shards are about to call the same contract. We call this attacking model triggered by intra-shard transactions as the *intra-shard attacking model*.

Except for the processing-execution difference between intra-shard and cross-shard transactions, there also exists a *processing-execution difference between cross-shard transactions from distinct shards*. To be more specific, shards have distinct consensus speeds and network transmission capabilities because of resource diversity, leading that transactions are processed and transferred across shards at various speeds. The resource diversity in a real-world scenario can include computing power, shard size, network bandwidth, etc. In this section, we will discuss a more flexible front-running attacking model based on such a difference in a sharded system, called *cross-shard attacking model*.

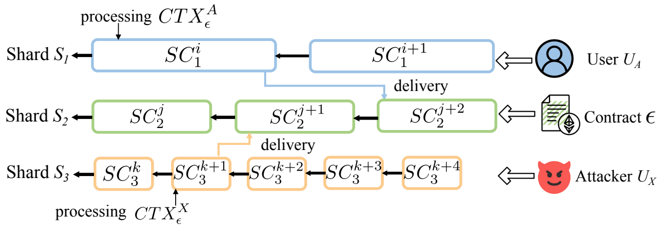


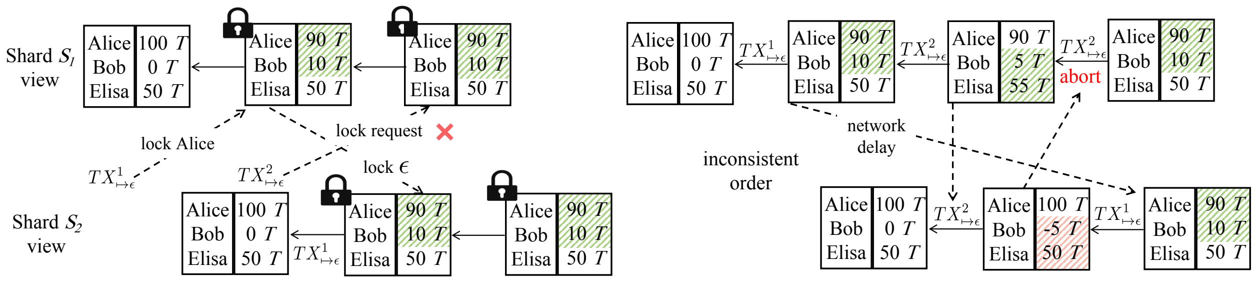
Fig. 21. The cross-shard attacking: the adversary registers accounts in a shard with faster consensus speed, where it utilizes the processing-execution difference between cross-shard transactions to launch a front-running attack.

Cross-shard attacking model. The adversary utilizes the processing-execution difference between cross-shard transactions from distinct shards to launch a front-running attack. Specifically, the adversary can gather in a shard that has the fastest consensus speed (i.e., the shortest block interval) and the highest bandwidth (i.e., the best network connection). By doing this, the attacker's cross-shard transactions will suffer a shorter processing time and delivery time, thus front-running to be executed by the contract's shard compared to the victims' cross-shard transactions.

Fig. 21 shows a concrete example of the cross-shard attacking model. We use the rectangle to represent a shard block and its length represents the consensus speed of this shard where a longer length means a longer time needed for consensus. Besides, from left to right of this figure is the increasing timestamp. We assume in this figure, shard S_1 maintains a victim's account U_A, S_2 maintains an order-sensitive contract ϵ, and S_3 maintains the attacker's account U_X. As shown in Fig. 21, S_3 can process transactions faster than S_1. In this case, once the attacker U_X monitors that a victim's cross-shard transaction (e.g., CTX_ϵ^A) is about to call ϵ, it immediately creates a cross-shard transaction (e.g., CTX_ϵ^X) in S_3. Since S_3 has faster consensus speed and shorter delivery network latency than S_1, the front-running transaction CTX_ϵ^X can be executed by S_2 even though CTX_ϵ^A is processed earlier (i.e., in block SC_1^i) than CTX_ϵ^X (i.e., in SC_3^{k+1}). Similarly, such a front-running attack can also happen to *every shard* that has a slower consensus speed or longer network latency than S_3.

Compared to the intra-shard attacking model, the adversary won't need to register multiple accounts in different contracts' shards with the cross-shard attacking model. For example, in the scenario of Fig. 21, the intra-shard attacking model asks the adversary to respectively register accounts in S_2 and other contracts' shards so that the adversary can construct intra-shard transactions to front-run the victim's cross-shard transactions. On the contrary, the adversary with the cross-shard attacking model only needs to register an account in the shard S_3 that has faster consensus speed and higher network bandwidth than other shards.

Haechi is immune to these attacking models. We find that these front-running attacking models rely on the processing-execution differences between intra-shard and cross-shard transactions or between cross-shard transactions from different shards. However, in Haechi, such processing-execution differences are eliminated since transactions calling the same contract are globally ordered before being executed via our fi-



(a) 2P cross-shard protocol: lock data to prevent data conflicts

(b) Optimistic cross-shard protocol: abort when existing data conflicts

Fig. 22. Different concurrency control mechanisms for handling cross-shard transactions.

nalization fairness algorithm proposed in section VI-B. Therefore, the execution order can be guaranteed consistent with the processing order, and Haechi ensures the finalization fairness.

Appendix B
Blocking and Non-blocking Processing

Sharding divides nodes into multiple shards to allow them to handle transactions in parallel. For intra-shard transactions from different shards, they can be processed and committed in parallel since they have *no data conflict*. However, for cross-shard transactions that may access the same data, a sharded system requires a concurrency control mechanism, i.e., cross-shard protocol, to ensure isolation and atomicity. For illustration, we assume two conflicting cross-shard transactions: (i) $TX^1_{\mapsto\epsilon}$: Alice in shard S_1 calls a smart contract ϵ in shard S_2 to transfer its 10 Tokens (denoted T) to Bob; (ii) $TX^2_{\mapsto\epsilon}$: Bob in shard S_1 calls the smart contract ϵ in shard S_2 to transfer its 5 T to Elisa. Note that Alice and Bob have their accounts maintained by S_1 while the token balances of Alice, Bob, and Elisa are a part of the contract state of ϵ maintained by S_2. Furthermore, we assume that Alice, Bob, and Elisa respectively have 100 T, 0 T, and 50 T at the beginning. Fig. 22 illustrates two concurrency control mechanisms for handling these two conflicting cross-shard transactions: 2P cross-shard protocol and optimistic cross-shard protocol.

Blocking processing. The 2P cross-shard protocol is a blocking processing mechanism where the processing of a later conflicting cross-shard transaction is blocked until the end of the cross-shard consensus of a former cross-shard transaction. The main idea of the 2P cross-shard protocol is to lock all relevant data to ensure isolation and atomicity when handling cross-shard transactions in parallel. It works well when two cross-shard transactions have no data conflicts, i.e., they don't have access to the same data and can be processed concurrently. However, it will block the processing of later conflicting cross-shard transactions if the previous cross-shard transaction keeps locking the conflicting data. As shown in Fig. 22(a), when S_1 and S_2 coordinately handle $TX^1_{\mapsto\epsilon}$, the contract state of ϵ is locked. If the conflicting $TX^2_{\mapsto\epsilon}$ hopes to request the lock to call ϵ, it will fail until the lock of ϵ is released by $TX^1_{\mapsto\epsilon}$. In this case, these two conflicting cross-shard transactions have to be handled one by one, i.e., blocking processing. Obviously, the blocking processing limits the capability of a sharded system in handling conflict cross-shard transactions. Especially in a real-world blockchain network, there are some popular smart contracts that are called frequently by users' transactions, and it is challenging to process these conflicting transactions well with the blocking processing mechanism.

Non-blocking processing. The optimistic cross-shard protocol can achieve non-blocking processing for cross-shard transactions. However, this protocol will introduce a high abortion rate when there are numerous conflicting transactions. As shown in Fig. 22(b), S_1 first handles $TX^1_{\mapsto\epsilon}$, and it *optimistically* assumes $TX^1_{\mapsto\epsilon}$ will be executed successfully, after which Bob will have 10 T in the contract ϵ. Then, S_1 starts processing $TX^2_{\mapsto\epsilon}$ without waiting for the final commitment of $TX^1_{\mapsto\epsilon}$. However, a network delay leads that $TX^2_{\mapsto\epsilon}$ arrives at S_2 before $TX^1_{\mapsto\epsilon}$. In this case, $TX^2_{\mapsto\epsilon}$ will be executed by S_2 before $TX^1_{\mapsto\epsilon}$. However, since the token balance of Bob is 0 T at that time, $TX^2_{\mapsto\epsilon}$ will lead to a negative value and thus fail to be executed. Finally, $TX^2_{\mapsto\epsilon}$ is aborted in both S_1 and S_2.

Comparing the two concurrency control mechanisms, we find that the optimistic cross-shard protocol is more suitable to be used in a practical scenario. Because there are always some popular smart contracts (e.g., DEXs discussed in § III-B) in a real-world blockchain network. The non-blocking processing property enables a sharded system to handle more transactions that call these popular smart contracts over time, without introducing a long-time lock waiting. However, the existing optimistic cross-shard protocol introduces frequent abortions if these transactions have numerous read/write conflicts for the same popular contracts. The reason why the optimistic cross-shard protocol leads to transaction abortions is that *conflicting transactions are out-of-order* before they are executed. To be more specific, since there is no global order for transactions that call the same contract, relevant shards may have different orders when processing these transactions. These inconsistent processing orders break the consistency and isolation of transactions and thus lead to transaction abortions.

Non-blocking processing in Haechi. Different from the existing optimistic cross-shard protocols, Haechi introduces an ordering phase to establish a global order of transactions for *each OSC*. By doing this, transactions from different shards calling the same OSC will have a *deterministic sequential order*. The deterministic order eliminates the concurrency conflicts and thus allows shards to independently and non-blocking process transactions. For those non-conflict contracts that do not interact with each other, Haechi can still handle their contract transactions in parallel. Therefore, compared to previous cross-shard protocols, Haechi can better process conflicting transactions in a non-blocking way and still support handling non-conflict transactions in parallel. This is promising to apply in a practical blockchain scenario.