Transaction Fairness in Blockchains, Revisited

Rujia Li, *IEEE Member,* Xuanwei Hu, *IEEE Student Member,* Qin Wang, *IEEE Member,*
Sisi Duan, *IEEE Senior Member,* Qi Wang, *IEEE Member*

Abstract—With the growing number of decentralized finance (DeFi) applications, transaction fairness in blockchains has gained much research interest. As a broad concept in distributed systems and blockchains, fairness has been used in different contexts, varying from ones related to the liveness of the system to ones that focus on the received order of transactions. In this work, we revisit the fairness definitions and find that existing fairness definitions are not adapted to blockchains with multiple DApps. We then provide a more generic one called *verifiable fairness*. Compared with prior definitions, our notion has two unique features: (i) it relaxes the ordering rules to a *predicate*, and thus gains in flexibility and generality; (ii) It enables users to independently verify if their transactions comply with the predicate for concrete applications. We also provide a scheme that achieves verifiable fairness, leveraging trusted hardware. Unlike prior works that usually design a dedicated consensus protocol to achieve fairness, our scheme can be integrated with any blockchain system. Our evaluation results on Amazon EC2 using up to 120 instances across different regions show that our construction imposes only minimal overhead on existing blockchain systems.

1 Introduction

Conventional consensus protocols [1], [2] in blockchains only guarantee that honest nodes reach an agreement on the order of transactions but do not care about the actual ordering of the transactions. With the growing interest in decentralized finance (DeFi), fairness of the transactions has gained research interest in recent years [3]–[8]. In DeFi applications, the lack of transaction fairness may lead to severe consequences, such as Maximal Extractable Value [9], transaction censorship [10], and consensus destabilization [11] (more damages refer to Appendix F).

Fairness in distributed systems and blockchains is a broad concept and a long-standing topic. It has been used in different contexts in the literature. For example, in 1978, Lamport first introduced *causal order* [12], where it claimed that a transaction that could have caused another transaction should always be committed first. In 2001, Cachin, Kursawe, Petzold, and Shoup introduced *block delivery fairness*. The notion requires that a transaction seen by a sufficiently large fraction of honest nodes will eventually be committed. This definition is directly related to the liveness of the protocol and is often embedded in the security properties of the protocol. Recently, in 2020, Kelkhar, Zhang, Goldfeder, and Jules introduced *order fairness*. It shows that the ordering of transactions received by a sufficiently large fraction of honest nodes should be preserved.

It is evident that the above definitions share two common principles. (1) The outcomes of each definition are deterministic, meaning that the ordering results can be verified by any honest nodes. (2) Each definition is distinctly crafted, with a precise focus on a specific consensus protocol or distributed system. For example, consider a scenario where block delivery is established as a fundamental rule, perceived as equitable. In this case, every transaction within the blockchain would adhere to this predetermined sequence. This method inherently relies on the premise that a single definition can be universally applicable across the entire system.

Dilemma of applying existing definitions to blockchains. Existing definitions face difficulties when implemented in real-world blockchain systems. Firstly, the ordering results in blockchain systems may not always be deterministic, and thus not verifiable. As summarized in the *result verifiability* column in Table 1, the order of transactions in 87% of our reviewed projects can not be verified. For instance, Dogecoin, Avalanche, and Monero order the transactions based on their *age*, i.e., the time a transaction enters a node's mempool. This makes it extremely challenging (if not impossible) to verify the ordering. Secondly, in the real-world setting, a blockchain may support a variety of applications, each with distinct transaction ordering needs. Take Ethereum as an example, where numerous diverse contract applications coexist, each demanding unique transaction sequencing. Specifically, in DeFi applications, the order of transactions is crucial due to the timing sensitivity of financial operations. Conversely, DID (Decentralized Identity) applications can operate with a more flexible ordering, like causal order, which prioritizes the relationships between transactions rather than their strict sequence. Therefore, it is challenging to devise a single fairness rule that can adequately satisfy the varied requirements of all these applications.

Rujia Li, and Sisi Duan are with Tsinghua University, Beijing, China; Email: {rujia, duansisi}@tsinghua.edu.cn.
Xuanwei Hu, and Qi Wang, are with Southern University of Science and Technology (SUSTech), Shenzhen, China. Email: {12232411,wangqi}@mail.sustech.edu.cn.
Qin Wang is with CSIRO Data61, Sydney, Australia. Email: qinwangtech@gmail.com.

TABLE 1: Transaction ordering in mainstream blockchain projects. *TxFee* orders the transactions according to the paid transaction fees. *Age* orders transactions based on the time they enter the mempool, sorted from the oldest to the newest. *Locality* prioritize transactions with local addresses. *Address* orders transactions by addresses; the transactions from the same address are ordered randomly. *Nonce* denotes a transaction counter for each sender's address. *TxHash* orders transactions by their hash values. *Valid time* refers to the period during which transactions remain valid; the transactions are discarded after the timer expires.

Project	Sequencing rules	Order algorithm	Verify algorithm	Result verifiability
Bitcoin Core	TxFee	miner.cpp:292	N/A	✗
Ethereum (Geth)	Locality + TxFee + Nonce	worker.go:1056	N/A	✗
Ripple	Address + Nonce + TxHash	CanonicalTXSet.cpp:25	Code	✓
Dogecoin	Age + TxFee	miner.cpp:425, 555	N/A	✗
Substrate	TxFee + Valid time	ready.rs:54	N/A	✗
LiteCoin	TxFee	miner.cpp:351	N/A	✗
Avalanche	Age	mempool.go:115, 226	N/A	✗
Monero	TxFee + Age	tx_pool.cpp:1484	N/A	✗
Tendermint	Priority + Age	mempool.go:297	N/A	✗
Ethereum Classic	Locality + TxFee + Nonce	worker.go:1074	N/A	✗
Stellar	Address(random) + Nonce	TxSetFrame.cpp: 408	N/A	✗
Bitcoin Unlimited	TxHash	miner.cpp: 268	Code	✓
Go-Algorand	Age	transactionPool.go	N/A	✗
Lotus	Locality + TxFee	selection.go: 42	N/A	✗
Thor	TxFee	tx_object.go: 95	N/A	✗

Our fairness definition. Fundamentally, the challenges mentioned above stem from the inapplicability of pre-set deterministic rules to specific applications. A natural solution is to allow each application to define its own ordering rules. This leads to our concept of *verifiable fairness*. Our definition does not aim to provide a new fairness rule. Instead, we introduce one *feature* unique to the fairness definition. First, our definition relaxes the sequencing rules that are embedded in prior fairness definitions. We generalize the sequencing rules to a *predicate* that is known to the public. Second, It empowers conventional fairness definitions with a new verifiability feature, where nodes can individually verify whether their transactions comply with the predicate. Under this definition, we can build one blockchain system accommodating the following scenarios: User A defines a time-reversed order for their DeFi application; User B defines a fee-based order for their DeFi application; User C defines an age-based order for their DeFi application. Each user – A, B, and C- can learn independently whether their transactions adhere to their specified rules in their DeFis.

Our fairness solution. We propose a solution that satisfies our new fairness definition. Instead of being a tailored solution that modifies the underlying consensus protocol, our method is a dedicated *ordering* protocol that can be integrated with any blockchain system. At the heart of our solution lies the utilization of a trusted execution environment (TEE) [13]–[15] to securely operate the mempool. Here, running a mempool inside a TEE brings several immediate benefits. First, sequencing algorithms are coded into TEEs, making them compatible with arbitrary rules. Second, our solution ensures that once a transaction enters the TEE, it is faithfully ordered according to predefined rules no matter who defines them. Third, transactions are encrypted and not revealed before they are proposed in a block, preventing front-running attacks at the stage of transaction submission.

While using a TEE-based mempool seems to be an intriguing idea to address the fairness issues, there are still several technical challenges when building a fully-fledged solution. Indeed, the TEE-based mempool is maintained by a blockchain node, also known as a TEE host. The TEE host is not assumed to be trustworthy and can behave arbitrarily. A malicious TEE host can still manipulate the order before transactions enter the mempool or simply discard the transactions. In fact, such behaviors cannot even be verified. We address the challenges by making the TEE host *accountable* for its misbehavior. In particular, by requesting a *receipt* from the TEE host, with overwhelming probability, a transaction sender can learn whether the TEE host misbehaves. We provide formal proof of our solution under the assumption that TEEs are trusted.

Practical contribution. We implement a functional prototype by leveraging the mainstream open-source project Go Ethereum (Geth) and OpenSGX [16]. To gauge the effectiveness of our solution, we evaluate its throughput and latency and compare it with Geth. Our evaluation with up to 120 nodes (deployed worldwide on AWS) shows that our TEE-based construction is only marginally slower than Geth. In particular, with 20 nodes, the throughput of our system is only 7% lower than that of Geth. With 120 nodes, the throughput of our system is 27% lower than that of Geth.

2 System Model and Preliminaries

Blockchain system model. We consider a blockchain system, an append-only ledger maintained by a group of nodes (also called miners or replicas). Let $\mathcal{N} = \{\mathrm{p}_0, \mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_{\iota-1}, \mathrm{p}_\iota\}$ be the set of $\iota+1$ nodes. A typical blockchain system works as follows. Initially, a client (e.g., denoted as c) creates a transaction tx and sends it to a blockchain node $\mathrm{p_i}$ $(0 \leq \mathrm{i} \leq \iota)$. Upon receiving a transaction tx, $\mathrm{p_i}$ adds it to its local mempool $\mathcal{P}_i$, where the mempool is a buffer of pending transactions. Then, nodes reach an agreement on the order of the transactions via a consensus protocol. In each epoch of the consensus protocol, nodes agree on one *block* of transactions. The block is usually proposed by some node. When the node creates a block proposer, it selects a set of transactions from its mempool and packs them into the block. After an agreement is reached, the transaction is *committed* on-chain.

We assume a computationally bounded adversary that can corrupt a fraction of nodes. A corrupt node behaves arbitrarily (also known as Byzantine failures [17]). We

also assume that all the nodes are rational: they seek to maximize their profits. We assume the blockchain system follows the standard assumption [18], which satisfies *consistency* and *liveness*. Generally, consistency ensures that honest nodes reach an agreement on the order of the transactions committed on-chain. The liveness property guarantees that all honest nodes will eventually reach an agreement on a valid transaction.

Trusted hardware. Trusted hardware is a broad term. This paper primarily focuses on the Trusted Execution Environment (TEE), a specialized and isolated hardware designed specifically for safeguarding sensitive code and data. Many TEE products such as Intel Software Guard Extensions (SGX) [13], ARM TrustZone [14], RISC-V Keystone [15] have been released. In this paper, we use Intel SGX as the TEE instance to illustrate TEE's key features: *runtime isolation* and *attestation*. Runtime isolation guarantees that the code execution is isolated from untrusted memories, while attestation is to prove that the application is running on trusted hardware. To capture the main functionalities of TEEs, we borrow the notion provided in prior works [19] and define the procedure of running algorithms within TEEs as a *black-box*. In particular, a TEE host first sets up a TEE and loads a program prgm into an enclave (secure and isolated execution unit). The program securely operates within such protected enclaves, ensuring the integrity of the execution. The enclave produces verifiable attestation to provide tangible evidence of the executed results. Subsequently, an external party can engage with an attestation service to validate the origin and authenticity of the generated outcomes. The abstraction for our trusted hardware HW is defined as follows.

- $\mathsf{HW.Setup}(1^\lambda)$: It takes as input a secure parameter λ. Upon setup, HW generates a key pair (sk_{quote}, vk_{quote}) for signing and verifying quotes, and public parameters pms for initializing enclaves.
- $\mathsf{HW.Init}(pms, \mathsf{prgm})$: It takes as input public parameters pms, a program prgm and outputs a program handle hdl. The handle corresponds to a running instance of the program. The instance state is stored by trusted hardware and can be retrieved via the handle.
- $\mathsf{HW.Run}(hdl, in)$: It runs the underlying instance of the program prgm via the handle hdl with an input in and outputs the execution result out.
- $\mathsf{HW.Run\&Quote}(hdl, in)$: It runs the program based on the handle hdl and input in and outputs a $quote = (hdl, tag_{\mathsf{prgm}}, in, out, \sigma)$, in which σ is a signature generated by sk_{quote} (regarding hdl), which proves that the result out is produced by hdl whose underlying program is prgm (identified by tag_{prgm}), with input in.
- $\mathsf{HW.VerifyQuote}(quote)$: This is the quote verification algorithm. It verifies whether a given $quote$ is valid by outputting b, where $b \in \{0, 1\}$. Here, 1 is on success or 0, and vice versa. In practice, this algorithm usually requires additional service (e.g., Intel Attestation Server for SGX users), and here we omit it for simplicity.

We assume HW achieves *execution integrity* and *remote-attestation-unforgeability* (formal definitions are provided in Appendix B). Informally speaking, the execution integrity property ensures that a correct program is executed. Meanwhile, the remote-attestation-unforgeability property is achieved if an adversary cannot (equiv. with a negligible probability) forge a quote passing the verification. We assume that the public parameters pms are known to all entities upon setup, and $\mathsf{HW.Run\&Quote}$ can always be invoked by the public parameters corresponding to $quote$.

Cryptographic primitives. Our system relies on standard cryptographic primitives such as public key encryption scheme PKE and signature scheme S. We provide the details of these algorithms in Appendix B.

3 Many Faces of Fairness Definitions

The notion of fairness has been used in different contexts in the literature of distributed systems and blockchain. We briefly summarize them into the following categories.

The notion was introduced by Reiter and Birman back in 1994 [20], later refined by Cachin, Kursawe, Petzold, and Shoup (CKPS) [21] and Duan, Reiter, and Zhang [22]. The causality relation requires that the system should preserve the standard notion of *causal order* introduced by Lamport [12]. Put it into the blockchain term, the definition is shown below.

Definition 1 (Causality)**.** *The transactions from an honest node should be committed in the order they are issued. If a transaction tx_1 from an honest node could have caused the transaction tx_2 from another honest node, tx_1 should be committed before tx_2.*

Reiter and Birman considered a scenario of trading stocks. When a node issues a transaction tx_1 to purchase stock shares, a faulty node may collude with other nodes to issue a derived request tx_2 for the same stock. If tx_2 is committed before tx_1, tx_2 may adjust the demand for the stock and the service may raise the price to the honest client. To preserve causality, the majority of existing solutions require that the transactions or proposals be encrypted so that the adversary cannot learn the contents to manipulate the order [20]–[22].

The notion of block delivery fairness was first formally introduced by CKPS [21] in 2001. The definition (rephrased) is shown below.

Definition 2 (Block delivery fairness)**.** *If the majority of honest nodes have received a transaction tx_1, then tx_1 is committed within a uniformly bounded time by honest nodes.*

The definition above is closely related to the liveness of the system. Namely, if an honest client submits a transaction to the system, the transaction will eventually

be committed, and the client will receive a reply. The term is sometimes used interchangeably with *censorship resilience* [23]. The block delivery fairness assures fairness between the nodes. To achieve the fairness goal, the protocol essentially needs to ensure that the proposal by honest nodes will eventually be committed within a bounded amount of time.

Kelkar, Zhang, Goldfeder, and Jules (KZGJ) [3] introduced the notion of order fairness, aiming to capture the order of transactions received by the nodes. In particular, if a sufficiently large fraction of nodes receive a transaction tx_1 before another transaction tx_2, tx_1 should be committed before tx_2. Unfortunately, this property cannot be achieved in practice due to the Condorcet paradox problem [24]. Namely, consider three nodes, X, Y, and Z, and three transactions a, b, and c. X receives them in the order [a, b, c], Y in the order [b, c, a], and Z in the order [c, a, b]. In this scenario, a majority of nodes have received (x before y), (y before z), and (z before x). This cyclic dependency makes it impossible to achieve order fairness. KZGJ thus relaxes the notion to *block order fairness*, as defined below.

Definition 3 (Block order fairness)**.** *If at least γ-fraction nodes receive a transaction tx_1 before tx_2, then the block that consists of tx_2 will not be committed before the block that consists of tx_1.*

The notion above allows tx_1 and tx_2 to be included in the same block, i.e., the transactions with cyclic dependencies are ordered *at the same height*. To achieve block order fairness, a tailored consensus protocol is proposed that involves all-to-all communication.

In a concurrent work, Kursawe [4] proposed *relative order fairness* and defined *timed relative fairness*. Timed relative fairness is weaker than block order fairness and thus the protocol can be much easier to build. The related definition is shown below.

Definition 4 (Timed relative fairness)**.** *If there is a time τ such that all honest nodes saw (according to their local clock) transaction tx_1 before τ and transaction tx_2 after τ, then tx_1 must be committed before tx_2.*

To achieve timed relative fairness, the protocol requires the nodes to maintain synchronized local clocks. All-to-all communication among the nodes is also required for the order in a proposed block to be validated. In another concurrent work, Zhang, Setty, Chen, Zhou, and Alvisi (ZSCZA) [5] proposed *ordering linearizability*.

Definition 5 (Ordering linearizability)**.** *If the highest timestamp that any node assigns to a transaction tx_1 is lower than the lowest timestamp that any honest node assigns to tx_2, then tx_1 is committed before tx_2.*

To achieve ordering linearizability, ZSCZA proposed a protocol that has a dedicated ordering phase. The phase involves two all-to-all communication steps. The order is determined according to the median timestamp included in two-thirds replicas, the value of which can be manipulated by an adversary [6].

Meanwhile, Cachin, Micic, Steinhauer, and Zanolini [7] refined the notion by KZGJ and proposed *differential order fairness*.

Definition 6 (Differential order fairness)**.** *Consider a system with $n \geq 3f+1$ nodes where f can be Byzantine (that fail arbitrarily). If the number of honest nodes that broadcast a transaction tx_1 before tx_2 exceeds the number that broadcast tx_2 before tx_1 by more than $2f+\kappa$, for some $\kappa \geq 0$, then the protocol must not commit tx_2 before tx_1.*

The protocol that achieves differential order fairness is much simplified compared to that by KZGJ, but still requires all-to-all communication.

Summary. It is evident that the *fairness* notion in existing definitions varies by context. In fact, building systems with a tailored fairness notion brings two drawbacks we seek to address in this work. First, existing definitions can only capture *deterministic* ordering rules. Unfortunately, this is often not the case in practical systems. For example, Dogecoin, Avalanche, and Monero order the transactions based on the time each transaction enters the mempool (i.e., *age*). There is no guarantee that the order of the transactions is deterministic. Second, to the best of our knowledge, all known solutions that achieve existing fairness notions are based on specific rules and require a dedicated (sometimes expensive) consensus protocol. However, in real-world systems, the rules are shaped by the requirements of higher-level applications, and existing rules may not simultaneously suit the needs of concrete applications running on the same blockchain.

4 A New Fairness Definition

In this section, we introduce a new fairness definition called *verifiable fairness*. The notion aims to be generic, capturing most fairness notions known so far. Building upon the notions of conventional fairness definitions, our definition has two advantages: (i) It allows fairness to be defined according to an arbitrary sequencing rule. (ii) It empowers conventional fairness definitions with a new verifiability feature, where nodes can individually verify whether their transactions comply with the rule.

4.1 Abstraction of Fairness Notion

We abstract away the notion of fairness and define a generic sequencing rule $\widetilde{R}$. The rule allows a node to determine the order of transactions based on the concrete application. We require that $\widetilde{R}$ is known by any nodes in the system. We define three algorithms for nodes to achieve fairness: Select, Order, and Verify. Select is queried by any node $\mathrm{p_i}$ (a prover) when it selects a batch of transactions from its mempool $\mathcal{P}_i$. Here, all transactions in $\mathcal{P}_i$ form a universal set called $\Gamma_{\mathcal{P}_i}$. The Order algorithm orders transactions. Verify allows any node (the verifier)

to verify whether the order of the transactions is valid. Formal definitions are shown below.

- $\mathsf{Select}(\Gamma_{\mathcal{P}_i})$: This algorithm is run by a prover that holds a mempool of transactions. The algorithm selects transactions from a transaction set $\Gamma_{\mathcal{P}_i}$ from its mempool, where we assume the size of $\Gamma_{\mathcal{P}_i}$ is ζ. The algorithm then takes as input $\Gamma_{\mathcal{P}_i}$ and outputs a transaction list $l = [tx_0, tx_1, \ldots, tx_{i-1}, tx_\tau]$. Ideally, $\zeta = \tau$, i.e., the algorithm selects all transactions in $\Gamma_{\mathcal{P}_i}$ and puts them into l.
- $\mathsf{Order}(l, ST, \widetilde{R})$: This algorithm is run by the prover. The algorithm takes as input a transaction list l (output of the Select algorithm), current blockchain state ST, a rule $\widetilde{R}$, and outputs an ordered sequence of transactions r.
- $\mathsf{Verify}(aux, r, \widetilde{R})$: This algorithm is run by a verifier who receives an ordered sequence of transactions r from the prover. The algorithm checks whether the transactions in r are sorted according to rule $\widetilde{R}$. It takes as input the auxiliary data aux, r, $\widetilde{R}$, and outputs *true* or *false*. Here, *true* means the prover orders the transactions in r according to $\widetilde{R}$.

Our abstraction of the algorithms can cover all the protocols we are aware of that achieve the fairness notions in Section 3. Indeed, all the protocols known so far hold an implicit assumption that *the verification of the ordered transactions by an honest node is consistent among all honest nodes*. Take the Aequitas protocol by KZGJ as an example, the protocol involves a gossip and broadcast phase dedicated for Order and Verify. The rule $\widetilde{R}$ is aligned with block order fairness in Definition 3. In particular, each node broadcasts the transactions in its mempool in the same order as they are received. In this way, each node builds the local state ST about available transactions and the order received by other nodes. Each node then proposes a batch of transactions using the Order algorithm based on the rule $\widetilde{R}$. Given the proposal by any node $\mathrm{p_i}$, nodes start a binary Byzantine agreement (BA) protocol [25], [26] to agree on whether a proposal should be committed. Namely, if an honest node receives the proposal from $\mathrm{p_i}$, it verifies whether the order is *valid*. If so, the node votes for 1 in BA and 0 otherwise. Finally, transactions in proposals where the corresponding BA outputs 1 are committed.

4.2 Defining Verifiable Fairness

In our definition, verifiable fairness means any node (transaction sender) can learn whether his transactions have been re-ordered or dropped (deviated from rules). Here, we define the transaction sender as the *verifier* and the transaction receiver as the *prover* (see Figure 1). Ideally, a verifier can check the transaction order without learning additional information from other nodes except the ordered result r. Consider that an ordering rule $\widetilde{R}$ is publicly available to all nodes. An ideal fairness notion should guarantee that a verifier can verify whether the prover has re-ordered the transaction set $\Gamma_{\mathcal{P}_i}$ or dropped any transactions (i.e., by not including the transactions in $\Gamma_{\mathcal{P}_i}$). Notably, here and in the rest of the paper, *re-order* refers to arbitrary deviations from the established rules in transaction sequencing. We define the following definition to formally capture our expectation for an ideal verifiable fairness definition.

Definition 7 (Ideal verifiable fairness, IV-fairness)**.** *A system achieves IV-fairness if the following holds. Given an arbitrary sequencing rule $\widetilde{R}$, the prover $\mathrm{p_i}$ queries $r \leftarrow \mathsf{Order}(l_i, ST, \widetilde{R})$ to obtain r, where $l_i \leftarrow \mathsf{Select}(\Gamma_{\mathcal{P}_i})$. The verifier can verify whether the prover has re-ordered or dropped transactions from $\Gamma_{\mathcal{P}_i}$ by querying $\mathsf{Verify}(\Gamma_{\star}, r, \widetilde{R})$. The* Verify *function returns true if $r \leftarrow \mathsf{Order}(l_i, ST, \widetilde{R})$.*

Unfortunately, ideal verifiable fairness is impractical. Determining whether a transaction has been dropped or re-ordered requires the verifier to possess knowledge of transactions in the prover's mempool $\mathcal{P}_i$. Unfortunately, the verifier may not be able to access this list, as the mempool is stored locally by the prover and may vary at different nodes (namely, $\Gamma_{\mathcal{P}_i} \neq \Gamma_{\mathcal{P}_j}$ for $i \neq j$).

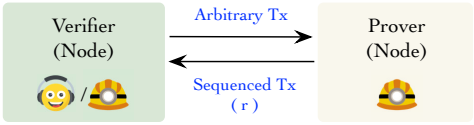


Fig. 1: Model sketch.

Then, we relax the definition and present another attempt. In this attempt, a verifier can submit an arbitrary set of transactions of its choice (e.g., l). We additionally require that the verification algorithm outputs a deterministic result (Appendix C) for a transaction-ordered list produced by $\mathsf{Order}(l, ST, \widetilde{R})$. The idea is that honest provers will always generate consistent verification results, regardless of the sequencing rules. It will thus be easy to build a protocol achieving fairness definitions while the ordering rules can be defined by concrete applications.

Attaining this property in practice is challenging. For instance, even if the sequencing rule incorporates non-deterministic elements, e.g. randomly sorting a transaction list, honest nodes may not generate consistent verification results. We thus continue to relax the definition and require each rule to be deterministic without any random elements, as in Definition 8.

Definition 8 (Deterministic verifiable fairness, DV-fairness)**.** *A blockchain system achieves DV-fairness if the following holds. Given a deterministic sequencing rule $\widetilde{R}$, a verifier provides a list of transactions l to the prover. The prover $\mathrm{p_i}$ queries $r \leftarrow \mathsf{Order}(l, ST, \widetilde{R})$ and returns r to the verifier. The verifier verifies whether the prover has re-ordered transactions in l or dropped transactions from l, by querying the $\mathsf{Verify}(l, r, \widetilde{R})$ function. The* Verify *function returns true*

if $r \leftarrow \mathsf{Order}(l, ST, \widetilde{R})$.

Unfortunately, such a definition is still not generic and cannot fully capture all the requirements. In real-world scenarios, the order of transactions from a prover often needs to be verified by multiple verifiers. It is thus too expensive for the prover to handle many verifiers concurrently, as the list l provided by the verifiers might be different. For instance, consider the scenario that the prover is already processing the request from a verifier $\mathrm{p_i}$ with list l as input. Another verifier $\mathrm{p_j}$ now submits another request with list $l' = l \cup \{m\}$ as input. The prover must also add m to its mempool to handle the request from $\mathrm{p_j}$. If the two requests are processed concurrently, the verification of the result of the order might not be correct anymore.

Our final attempt seeks to achieve a trade-off between Definition 7 and Definition 8. In this new attempt, the list of transactions l is not provided by the verifier anymore. Instead, the prover selects and orders transactions based on its mempool. This change requires each verifier to verify independently whether the prover intentionally drops or re-orders its transactions. The verification of other verifiers is irrelevant.

Definition 9 (Individual verifiable fairness, IDV-fairness)**.** *A blockchain system achieves IDV-fairness if the following holds. Given an arbitrary sequencing rule* $\widetilde{R}$*, the prover queries* $r \leftarrow \mathsf{Order}(l_i, ST, \widetilde{R})$ *to obtain* r*, where* $l_i \leftarrow \mathsf{Select}(\Gamma_{\mathcal{P}_i})$*. A verifier can verify whether its transaction* tx *has been re-ordered or dropped from* r *by querying* $\mathsf{Verify}(tx, r, \widetilde{R})$*. The* Verify *function returns true if* $r \leftarrow \mathsf{Order}(l_i, ST, \widetilde{R})$ *where* $tx \in \Gamma_{\mathcal{P}_i}$*.*

Our IDV-fairness covers most fairness notions known so far, especially order-related fairness definitions. Namely, given an arbitrary rule $\widetilde{R}$, any honest client can learn whether a node has faithfully ordered transactions according to the rule.

5 Secure Construction in IDV-fairness

In this section, we propose a construction that satisfies IDV-fairness. Our construction is a dedicated protocol for ordering transactions, leveraging TEEs. It aims to make a verifier learn whether the prover has re-ordered or dropped his transactions. We first delve into the technical challenges and then present our proposed approach.

Threat model. We require blockchain nodes to be equipped with TEEs, and assume TEEs are secure. Also, we follow the standard assumption of the blockchain system, where a threshold number of blockchain nodes is Byzantine, and these Byzantine nodes (TEE hosts) are not trusted and can behave maliciously.

Technical challenges. A native solution can be built as follows. The ordering rule $\widetilde{R}$ is first compiled as a program and loaded into a TEE, where the TEE hosts the mempool $\mathcal{P}_i$ for each node. Then, a client c (a verifier) encrypts a transaction tx and obtains the ciphertext ct_{tx}, and sends ct_{tx} to a node $\mathrm{p_i}$ (a prover). Then, $\mathrm{p_i}$ broadcasts ct_{tx} to other nodes and meanwhile transfers ct_{tx} to its mempool $\mathcal{P}_i$. Next, $\mathcal{P}_i$ decrypts ct_{tx}, obtains tx, and appends tx to its secure memory. After that, $\mathcal{P}_i$ signs tx as a receipt rep_{tx}, and returns rep_{tx} to the client. The transaction tx is later processed on-chain according to the specification of blockchain. The transaction tx is removed from the mempool if tx is committed on-chain.

Intuitively, the above approach already allows the client to verify whether its transaction tx has entered a TEE-based mempool and is correctly ordered. Unfortunately, there are still a few challenges when building a provably secure approach, even assuming TEEs are fully trusted. We consider two cases: after c submits tx to $\mathrm{p_i}$, $\mathrm{p_i}$ fails to return rep_{tx} within a bounded amount of time; and $\mathrm{p_i}$ returns rep_{tx}. Verifiability in the first case is straightforward. Indeed, if rep_{tx} is not received, the TEE host misbehaves, e.g., the TEE host fails to transfer tx to its mempool or it has not broadcast tx to other nodes.

The second case is much trickier. Even if a TEE host returns rep_{tx} to c, it does not necessarily follow the specification of the protocol. In particular, a malicious TEE host may (i) not broadcast ct_{tx} to other nodes; (ii) maliciously drop tx after tx has been added to the mempool. In this way, the TEE host *tricks* the TEE to generate a correct receipt and then disobey the protocol, so tx might never be committed on-chain.

In fact, if tx is submitted to only one node $\mathrm{p_i}$ in a blockchain system, the fact that tx might never been committed on-chain is already a challenge in blockchain systems that is impossible to address, regardless of whether TEE-based mempool is used or not. Indeed, as $\mathrm{p_i}$ might be malicious, it can simply drop tx so tx will never be processed on-chain. In practice, the client c can simply send tx to another node. If one honest node receives tx, the liveness property of the blockchain ensures that tx will eventually be committed on-chain.

Our solution takes a step further by reducing the second case to the first case. In particular, we provide two crucial components: *trusted broadcast* and *trusted refresh*. Trusted broadcast requires that a TEE-based mempool only accepts a transaction tx only after the TEE host has provided proof that tx has been sent to a set of randomly selected nodes by the TEE. As the set of nodes is randomly selected by the TEE, with high probability, at least one honest node has received tx. A nice feature of this approach is that after c receives the receipt rep_{tx}, tx will eventually be committed on-chain. Meanwhile, trusted refresh requires that the TEE host provides a piece of evidence that tx has been committed before tx is removed from the mempool. By using trusted broadcast and trusted refresh altogether, we know that if the client does not receive a receipt, the TEE host must misbehave.

5.1 Overview of the Construction

At a high level, our construction works as follows (as illustrated in Figure 2). Initially, nodes set up their TEEs and load mempool into the TEEs. Then, a client c encrypts transaction tx and obtain the ciphertext ct_{tx}, and then sends ct_{tx} to a node p_i. ① Upon receiving ct_{tx}, p_i selects some peers which are randomly supplied by the TEE, and then sends ct_{tx} to these peers. When a node receives such a request, it returns a receipt com_{tx} to p_i where com_{tx} can simply be a digital signature. ② When p_i receives receipts com_{tx} from a sufficiently large number of peers, p_i forwards ct_{tx} and the receipts to the mempool $\mathcal{P}_i$. If com_{tx} is valid, $\mathcal{P}_i$ accepts ct_{tx}. ③ After $\mathcal{P}_i$ accepts ct_{tx}, the TEE decrypts ct_{tx} and appends tx to the mempool. Then $\mathcal{P}_i$ generates a signature (a receipt rep_{tx}) on tx. Here, rep_{tx} is used to prove that tx has been included in the mempool and has been broadcast to other nodes. ④ When p_i needs to select a set of transactions from the mempool (e.g., when preparing a block proposal), it queries $\mathcal{P}_i$. $\mathcal{P}_i$ orders the transactions and obtains a list of ordered transactions r. The TEE also creates an attestation σ_r for the verification of the order in r. Upon receiving the block proposed by p_i, other honest nodes verify whether the order is valid by verifying the attestation σ_r. ⑤ After tx is committed on-chain, p_i queries $\mathcal{P}_i$ to remove tx. $\mathcal{P}_i$ verifies whether tx has indeed been committed on-chain (the verification algorithm depends on the concrete blockchain system). Finally, $\mathcal{P}_i$ removes tx from the mempool. Notably, steps ①- ② are procedures for trusted broadcast, and ⑤ is the procedure for trusted refresh.

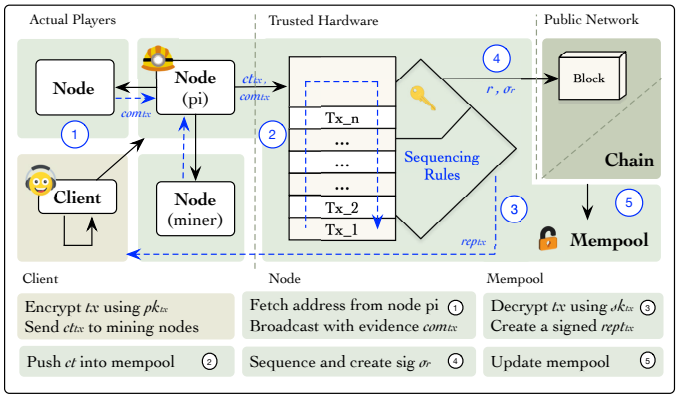


Fig. 2: System overview.

Our solution satisfies Definition 9. First, the sequencing rule is left as a predicate to the system. Indeed, the sequencing rule is programmed into TEE, so any rule can be implemented. Second, any verifier c can verify whether his transaction tx has been re-ordered or dropped. To be specific, if p_i does not return a receipt rep_{tx} to c within a bounded amount of time, c knows that p_i misbehaves. On the contrary, if p_i returns the receipt rep_{tx} to c, c knows that tx will eventually be committed on-chain. Namely, after c receives the receipt, tx must have been broadcast to a sufficiently large number of nodes in the system. Since the list of nodes is randomly selected by the TEE, the only thing left is to ensure that the list is large enough so that at least one honest node receives tx. The liveness property of the blockchain thus ensures that tx will eventually be committed on-chain.

Our approach enjoys two immediate benefits. First, our approach provides a solution where the re-ordering or dropping of transactions can be detected and verified. In blockchain systems, transaction fairness is involved in multiple procedures (see Table 2). As transaction ordering is usually under the control of individual nodes, one malicious node can re-order or drop transactions. Furthermore, other nodes are unable to differentiate this situation from that the transaction is simply not received. Our approach thus takes a step further by allowing the client to be aware of whether its transactions have been re-ordered or dropped.

TABLE 2: Comparison of Ethereum and our construction.

		Ethereum		Ours	
		Prevention	*Awareness*	*Prevention*	*Awareness*
Adversarial	① p_i rejects to broadcast tx	✗	✗	✗	✓
	② p_i rejects to send tx into $\mathcal{P}_i$	✗	✗	✗	✓
	③ p_i maliciously re-orders tx	✗	✗	✓	✓
	⑤ p_i maliciously removes tx from $\mathcal{P}_i$	✗	✗	✓	✓
Honest	① p_i faithfully broadcasts tx	-	✗	-	✓
	② p_i faithfully sends tx into $\mathcal{P}_i$	-	✗	-	✓
	③ p_i faithfully orders tx	-	✗	-	✓
	⑤ p_i faithfully removes tx from $\mathcal{P}_i$	-	✗	-	✓

Second, our solution can be integrated with any blockchain system, gaining flexibility (of the sequencing rule) and modularity. Our TEE-based construction is an independent protocol that can be integrated with any existing blockchain systems we are aware of. We briefly discuss the compatibility with Ethereum 2.0 (Eth2) as an example. Eth2 now adopts Proof-of-Stake (PoS) as the consensus mechanism. With PoS, each node (equiv. validator) takes a certain amount of tokens to be able to vote. Validators order the transactions in their block proposals. To use our solution, validators need to install TEE-based mempools. Before a validator proposes a block, it selects the transactions from its TEE-based mempools. Other validators vote for the block only if the order is verified.

Meanwhile, our solution is compatible with the different blockchain architectures today, as shown in Figure 3. Existing architectures can be categorized into three types. The first type involves a direct connection between users and the validator's mempool. The second type uses synchronized nodes as intermediate relays to reduce the workloads of the validators. The last type relies on MEV-boost-relays to separate the packing of transactions (used for block proposals) from actual block

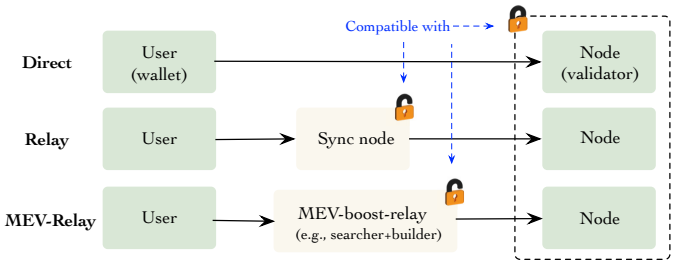


Fig. 3: Typical blockchain architectures.

proposals [27]. Our solution can be applied to all these architectures, replacing the mempool functions at the validators, sync nodes, and MEV-relays in the three architectures, respectively. In fact, our solution can make also MEV-relays accountable, which is a desirable feature for today's MEV [28].

5.2 Detailed Protocol

The protocol consists of two major stages: the *setup* stage and the *interaction* stage. The pseudocode is shown in Figure 4. Without loss of generality and for ease of presentation, when we present the trusted broadcast, the prover only needs to obtain one receipt from other nodes. In practice, this can be set up as a system parameter.

Setup stage. The setup stage (or S in short) aims to initialize the TEE-based mempool.

S1: Rule establishing. This stage establishes the sequence rule $\widetilde{R}$ for ordering transactions. $\widetilde{R}$ defines transaction admission, eviction, ordering, and dropping (customized by algorithms Select and Order in Section 4.1). Next, nodes compile $\widetilde{R}$ into a program and load them into TEE their enclaves.

S2: Key negotiation. Two key pairs are set up for each TEE. As TEE key management is not the main focus of our work, we assume that all TEEs share the same key. First, an arbitrary node p_0 is selected as an initial node, whose mempool $\mathcal{P}_0$ is responsible for generating the key pairs $(pk_{\mathrm{tx}}, sk_{\mathrm{tx}})$ and $(vk_{\mathrm{list}}, sk_{\mathrm{list}})$ by calling HW.Run&Quote$(hdl_{\mathcal{P}_0}, \text{“InitKeyGen”})$. Next, any other node $\mathrm{p_i}$ $(0 < \mathrm{i} \leq \iota)$ initializes its mempool $\mathcal{P}_i$ in the TEE. In particular, the TEE fetches the key pairs from $\mathcal{P}_0$ via remote attestation: $\mathcal{P}_i$ generates a temporary public-private key pair $(pk_{\mathcal{P}_i}, sk_{\mathcal{P}_i})$ and a $quote$ by calling HW.Run&Quote$(hdl_{\mathcal{P}_i}, \text{“TempKeyGen”})$. After that, $\mathcal{P}_i$ sends $pk_{\mathcal{P}_i}$ and $quote$ to $\mathcal{P}_0$. After receiving the message, p_0 calls HW.Run&Quote$(hdl_{\mathcal{P}_0}, (\text{“KeyEnc”}, pk, quote))$ in its TEE. The TEE (i) confirms that $pk_{\mathcal{P}_i}$ indeed comes from a valid enclave by verifying $quote$, and (ii) confirms that $\mathrm{p_i}$ is a valid node, and (iii) encrypts sk_{list} and sk_{tx} as ct_k using $pk_{\mathcal{P}_i}$. (lines 41-48 in Figure 4). Finally, p_0 sends ct_k to $\mathrm{p_i}$, which transfers ct_k to $\mathcal{P}_i$. $\mathcal{P}_i$ decrypts ct_k and stores sk_{list}, sk_{tx} in its local memory.

S3: Peer setup. p_0 retrieves a list of IP addresses in the network from the bootnode. Then, these addresses are loaded into enclaves via remote attestation.

Interaction stage. There are five sub-algorithms in this stage (T in short), shown as follows.

T1: Tx submission. Client c verifies pk_{tx} by querying HW.Verify$(quote_{\mathrm{pk}})$. This ensures that pk_{tx} is indeed generated in a valid enclave. Then, c uses pk_{tx} to encrypt a transaction tx and sends corresponding ct_{tx} to $\mathrm{p_i}$.

T2: Trusted broadcast. Upon receiving ct_{tx}, $\mathrm{p_i}$ calls $\mathcal{P}_i$ to fetch a list of addresses. Here, $\mathcal{P}_i$ randomly selects a set of nodes from $\mathcal{N}$ and returns the addresses of the nodes to the TEE host. Then, $\mathrm{p_i}$ sends ct_{tx} to the chosen addresses. Upon receiving such a message, each node creates a signature and sends a confirmation message com_{tx} to $\mathrm{p_i}$. After $\mathrm{p_i}$ collects at least one confirmation com_{tx} from one of the selected nodes, $\mathrm{p_i}$ transfers com_{tx} and ct_{tx} to its mempool $\mathcal{P}_i$.

T3:Tx merge. $\mathrm{p_i}$ calls HW.Run$(hdl_{\mathcal{P}_i}, (\text{“Merge”}, ct_{tx}))$ to perform the following operations (lines 59-63): (i) it verifies com_{tx}; (ii) it verifies the signature of tx. If the verification succeeds, the following steps are executed sequentially: (iii) $\mathcal{P}_i$ decrypts ct_{tx} using sk_{tx} to obtain the transaction tx; (iv) $\mathrm{p_i}$ appends tx to its mempool, i.e., $\Gamma_{\mathcal{P}_i} = \Gamma_{\mathcal{P}_i} \bigcup \{tx\}$; (v) $\mathrm{p_i}$ generates a receipt rep_{tx}, proving that tx has been merged into $\Gamma_{\mathcal{P}_i}$. Here, the proof rep_{tx} is in the form of $(\mathsf{H}(tx), \sigma)$, where σ is a signature of tx. Next, $\mathcal{P}_i$ sends the receipt rep_{tx} to c.

T4: Ordering & consensus. When $\mathrm{p_i}$ needs to order the transactions (e.g., during block proposal), it queries HW.Run$(hdl_{\mathcal{P}_i}, \text{“TxRetrieval”})$ (lines 64-70) to perform the following operations. (i) $\mathcal{P}_i$ queries Select and Order to obtain a sorted transaction list r. (ii) $\mathcal{P}_i$ creates a signature for r using its secret key sk_{list} and obtains a signature σ_r. Here, σ_r proves that r is generated and ordered by a secure mempool. (iii) $\mathcal{P}_i$ returns r and σ_r to $\mathrm{p_i}$. Also, $\mathcal{P}_i$ sets the variable $ordered$ and rejects all the following order requests from p_i, before new transactions are committed on the blockchain. This prevents p_i from repeatedly querying its secure mempool to obtain biased ordered transactions. Now, if $\mathrm{p_i}$ needs to create a block proposer, it adds r to its block. When it broadcasts the block, it includes σ_r in the block as well. Other nodes can verify whether the sorted transaction list r is generated according to the rule $\widetilde{R}$ by querying whether S.Verify$_{vk_{\mathrm{list}}}(r, \sigma_r)$ returns 1.

T5: Trusted refresh. Transactions committed on-chain must be removed from the mempools. When a new block is committed, $\mathrm{p_i}$ sends the block to its mempool $\mathcal{P}_i$. Meanwhile, a piece of evidence that the block is committed is also provided to $\mathcal{P}_i$. Upon receiving the block and a proof, $\mathcal{P}_i$ checks the proof, and removes the corresponding transactions from its TEE-based mempool. In permissioned blockchains, proof can involve signatures from a consensus node quorum. For instance, in Proof-of-Work (PoW)-based blockchains, the

Mempool Setup

Node p_0: Initialization

1: $pms_0 \leftarrow \mathsf{HW.Setup}(1^\lambda)$
2: $hdl_{\mathcal{P}_0} \leftarrow \mathsf{HW.Init}(pms_0, P)$
3: $(pk_{\text{tx}}, vk_{\text{list}}), quote_{\text{pk}} \leftarrow \mathsf{HW.Run\&Quote}(hdl_{\mathcal{P}_0}, \text{“InitKeyGen”})$

Node p_i $(i > 0)$: Initialization

4: $pms_i \leftarrow \mathsf{HW.Setup}(1^\lambda)$
5: $hdl_{\mathcal{P}_i} \leftarrow \mathsf{HW.Init}(pms_i, P)$
6: $pk_{\mathcal{P}_i}, quote_{\mathcal{P}_i} \leftarrow \mathsf{HW.Run\&Quote}(hdl_{\mathcal{P}_i}, \text{“TempKeyGen”})$
7: **Send** $pk_{\mathcal{P}_i}$ and $quote$ to p_0

Node p_0: KeyEncryption

8: **Receive** $pk_{\mathcal{P}_i}$ and $quote_{\mathcal{P}_i}$ from p_i
9: $ct_k, quote_{ct_k} \leftarrow \mathsf{HW.Run\&Quote}(hdl_{\mathcal{P}_0}, (\text{“KeyEnc”}, pk_{\mathcal{P}_i}, quote_{\mathcal{P}_i}))$
10: **Send** ct_k and $quote_{ct_k}$ to p_i

Node p_i $(i > 0)$: KeyDecryption

11: **Receive** ct_k and $quote_{ct_k}$ from p_0
12: $\perp \leftarrow \mathsf{HW.Run}(hdl_{\mathcal{P}_i}, (\text{“KeyDec”}, ct_k, quote_{ct_k}))$

Transaction Submission

Node c: TxEncryption

13: **Receive** $quote_{\text{pk}}$ from a node p_j
14: **Assert** $\mathsf{HW.VerifyQuote}(quote_{\text{pk}}) = 1$
15: Parse $quote_{\text{pk}} = (hdl_{\mathcal{P}_0}, tag_{\mathcal{P}_0}, in, out, \sigma)$
16: **Assert** $tag_{\mathcal{P}_0} = tag_P$ **and** $in =$ “InitKeyGen” **and** $out = (pk_{\text{tx}}, \cdot)$
17: $ct_{tx} \leftarrow \mathsf{PKE.Enc}_{pk_{\text{tx}}}(tx)$
18: **Send** ct_{tx} to a node p_i

Node p_i: TxBroadcast

19: **Receive** ct_{tx} from c
20: $(\mathrm{pr}_0, \ldots, \mathrm{pr}_a) \leftarrow \mathsf{HW.Run}(hdl, \text{“GetPeers”})$
21: **Send** ct_{tx} to peers $\mathrm{pr}_0, \ldots, \mathrm{pr}_a$

Node pr_i: ReceiveConfirmation

22: **Receive** ct_{tx} from p_i
23: $com^i_{tx} \leftarrow \mathsf{S.Sign}_{sk_{\mathrm{pr}_i}}(ct_{tx})$
24: **Send** com^i_{tx} to p_i

Node p_i: TxMerge

25: **Receive** $com_{tx} := (com^0_{tx}, \ldots, com^{a'}_{tx})$ from remote peers
26: $rep_{tx} \leftarrow \mathsf{HW.Run}(hdl_{\mathcal{P}_i}, (\text{“Merge”}, ct_{tx}, com_{tx}))$
27: **Send** rep_{tx} to c

Transaction Commitment

Node p_i: BlockProposing

28: $r, \sigma_r \leftarrow \mathsf{HW.Run}(hdl_{\mathcal{P}_i}, \text{“TxRetrieval”})$
29: Propose a block B with transaction list r
30: **Broadcast** block B along with signature σ_r

Node p_j: BlockVerification

31: **Receive** block B and signature σ_r. Extract transaction list r from B.
32: Accept block B only if
33: $\mathsf{S.Verify}_{vk_{\text{list}}}(r, \sigma_r) = 1$

Enclave Calls

procedure $\mathsf{HW.Run\&Quote}(hdl, \text{“InitKeyGen”})$

34: $pk_{\text{tx}}, sk_{\text{tx}} \leftarrow \mathsf{PKE.Gen}(1^\lambda)$
35: $vk_{\text{list}}, sk_{\text{list}} \leftarrow \mathsf{S.Gen}(1^\lambda)$
36: Generate $quote_{\text{pk}}$ to prove the correct generation of the public-private key pairs
37: **Return** $(pk_{\text{tx}}, vk_{\text{list}}), quote_{\text{pk}}$

procedure $\mathsf{HW.Run\&Quote}(hdl, \text{“TempKeyGen”})$

38: $pk, sk \leftarrow \mathsf{PKE.Gen}(1^\lambda)$
39: Generate $quote$ to prove the correct generation of the public-private key pair
40: **Return** $pk, quote$

procedure $\mathsf{HW.Run\&Quote}(hdl, (\text{“KeyEnc”}, pk, quote))$

41: $b := \mathsf{HW.VerifyQuote}(quote)$
42: **If** $b = 0$ **then**
43: **Return** $\perp$
44: Parse $quote = (hdl, tag, in, out, \sigma)$
45: **If not** $(tag = tag_P$ **and** $in =$ “TempKeyGen” **and** $out = pk)$ **then**
46: **Return** $\perp$
47: $m := (pk_{\text{tx}}, sk_{\text{tx}}, vk_{\text{list}}, sk_{\text{list}})$
48: $ct_k \leftarrow \mathsf{PKE.Enc}_{pk}(m)$
49: Generate $quote_{ct_k}$ to prove the correct encryption of the key pairs
50: **Return** $ct_k, quote_{ct_k}$

procedure $\mathsf{HW.Run}(hdl, (\text{“KeyDec”}, ct_k, quote_{ct_k}))$

51: $b := \mathsf{HW.VerifyQuote}(quote_{ct_k})$
52: **If** $b = 0$ **then**
53: **Return** $\perp$
54: Parse $quote = (hdl, tag, in, out, \sigma)$
55: **If not** $(tag = tag_P$ **and** $in =$ “KeyEnc” **and** $out = ct_k)$ **then**
56: **Return** $\perp$
57: $(pk_{\text{tx}}, sk_{\text{tx}}, vk_{\text{list}}, sk_{\text{list}}) \leftarrow \mathsf{PKE.Dec}_{sk}(ct_k)$
58: **Return**

procedure $\mathsf{HW.Run}(hdl, (\text{“Merge”}, ct_{tx}, com_{tx}))$

59: **If** no receipt in com_{tx} is valid **or** $tx \in \Gamma$ **then**
60: **Return** $\perp$
61: $tx := \mathsf{PKE.Dec}_{sk_{\text{tx}}}(ct_{tx})$
61: $\Gamma := \Gamma \cup \{tx\}$
62: $\sigma \leftarrow \mathsf{S.Sign}_{sk_{\text{list}}}(\mathsf{H}(tx))$ ▷ $\mathsf{H}(tx)$ denotes the hash of the transaction
63: **Return** $\mathsf{H}(tx), \sigma$

procedure $\mathsf{HW.Run}(hdl, \text{“TxRetrieval”})$

64: **If** $ordered = true$ **then** ▷ $ordered$ indicates whether transactions have been ordered or not, initially set to $false$
65: **Return** $\perp$
66: $l \leftarrow \mathsf{Select}(\Gamma)$
67: $r \leftarrow \mathsf{Order}(l, ST, \widetilde{R})$
68: $\sigma_r \leftarrow \mathsf{S.Sign}_{sk_{\text{list}}}(r)$
69: $ordered := true$
70: **Return** r, σ_r

procedure $\mathsf{HW.Run}(hdl, (\text{“TxUpdate”}, B))$

71: **If** B is invalid **then**
72: **Return** $\perp$
73: Retrieve the list of committed transactions l in B
74: **For each** transaction tx in l **do**
75: $\Gamma := \Gamma \backslash \{tx\}$
76: $orderd := false$
77: **Return**

procedure $\mathsf{HW.Run}(hdl, \text{“GetPeers”})$

78: Randomly select peers $(\mathrm{pr}_0, \ldots, \mathrm{pr}_a)$ from $\mathrm{p}_0, \ldots, \mathrm{p}_\iota$
79: **Return** $\mathrm{pr}_0, \ldots, \mathrm{pr}_a$

Fig. 4: Our protocol that satisfies IDV-fairness.

proof can be achieved by adopting a proof-of-publication protocol as mentioned in [29].

Notably, if TEEs are compromised, the system fails to achieve IDV-fairness. However, other properties of the blockchain (i.e., consistency and liveness) are not violated (see our discussion in Appendix D).

6 Security Analysis

Theorem 1. *If* HW *satisfies the security requirements outlined in Definition 12 and attains remote attestation unforgeability, as specified in Definition 13, and provided that* PKE *offers IND-CCA2 security while* S *is EUF-CMA secure, there exists a negligible probability that an honest verifier fails to detect that an adversarial node re-orders the transactions submitted by that verifier.*

Proof. We prove this theorem by contradiction. Assuming that an adversarial node $\mathcal{A}$ can re-order a transaction tx sent by an honest verifier, then we can use $\mathcal{A}$'s abilities to break our assumptions. The fact that $\mathcal{A}$ successfully re-orders tx without being noticed indicates the occurrence of two events: tx has been re-ordered; a valid rep_{tx} is generated. We define them as $\mathbb{E}_{\mathcal{A}}^{\text{re-order}}$, and $\mathbb{E}_{\mathcal{A}}^{\text{receipt}}$, respectively. We first examine probability of that $\mathbb{E}_{\mathcal{A}}^{\text{re-order}}$ occurs. The sequence result is derived from the execution of Order and Select. The fact $\mathbb{E}_{\mathcal{A}}^{\text{re-order}}$ occurs implies that a compromised TEE exists. In particular, $\mathcal{A}$ has manipulated the execution of either Select or Order, a violation of the execution integrity property of the TEE, as defined in Definition 12. Namely, $\Pr\left[\mathbb{E}_{\mathcal{A}}^{\text{re-order}}\right] \leq \Pr\left[\mathsf{Exec\text{-}integrity}_{\mathcal{A},\mathsf{HW}}(\lambda) = 1\right]$.

We then examine the probability of that $\mathbb{E}_{\mathcal{A}}^{\text{receipt}}$ occurs. In our solution, rep_{tx} represents a signature on tx. We consider two possible cases: (Case-1) the private key was not compromised; (Case-2) the private key was compromised.

Case-1: $\mathcal{A}$ does not possess the private key. However, $\mathcal{A}$ generates a new signature, a violation of the EUF-CMA security defined in Definition 11.

Case-2: The private key has been compromised. Here, there are two sub-cases. First, $\mathcal{A}$ interacts with a mempool $\mathcal{P}_j$ that has been successfully established. To make $\mathcal{P}_j$ transmit the key pairs to $\mathcal{A}$, $\mathcal{A}$ must forge the quote $quote_{\mathcal{P}_i}$ such that $tag_{\mathcal{P}_i} = tag_{\mathsf{prgm}}$. This action violates the unforgeability of remote attestation, as specified in Definition 13. Second, $\mathcal{A}$ does not interact with the mempool. In this scenario, $\mathcal{A}$ acquires knowledge of the key pairs from the ciphertext transmitted between the mempools. This violation corresponds to IND-CCA2 security of PKE scheme, as described in Definition 10.

Thus, the probability that $\mathcal{A}$ re-orders transaction tx sent by an honest verifier while not detected, is negligible. □

Theorem 2. *If* HW *satisfies the security requirements outlined in Definition 12, and provided that* S *is EUF-CMA secure, there exists a negligible probability that an honest verifier will fail to detect that an adversarial node drops the transactions submitted by that verifier.*

Proof (sketch)**.** Suppose that an honest verifier fails to detect an adversarial node $\mathcal{A}$ dropping its transaction, then we can transform $\mathcal{A}$'s ability to generate a valid receipt rep_{tx} for the verifier. We define this event as $\mathbb{E}_{\mathcal{A}}^{\text{drop}}$. When a verifier receives a valid receipt rep_{tx} for transaction tx, $\mathcal{P}_i$ must have had accepted com_{tx}. From line 59 in Figure 4, we know that $\mathcal{P}_i$ accepts com_{tx} only if its signature is successfully verified. Thus, to make com_{tx} be accepted, $\mathcal{A}$ must either forge a signature or manipulate the programs running in a TEE. However, the former case violates the unforgeability of a secure signature, while the second case breaks the execution integrity of TEE as defined in Definition 12. The probability that any case occurs is negligible. Thus, we conclude that $\Pr[\mathbb{E}_{\mathcal{A}}^{\text{drop}}]$, the probability that the verifier fails to detect that its transaction is dropped by $\mathcal{A}$, is negligible.

7 Implementation

We build a prototype[1] in Golang and C. We use Go Ethereum (Geth) as the blockchain. For TEE, we use both OpenSGX [16] as the TEE instance and Intel SGX SDK. Geth is the most widely used Golang implementation of Ethereum while OpenSGX emulates Intel SGX. To implement our system architecture shown in Figure 2, we use different modules in Geth for different purposes. In particular, we use the miner module and the worker module as the nodes in our protocol, and extend the API module to implement the clients. For the consensus protocol, we use Clique in the consensus module of Geth. We split the mempool module from Geth into an independent process and disable the corresponding functionalities from the original mempool module in Geth. We run the mempool inside SGX to implement our TEE-based mempool. We specify the sequencing rule $\widetilde{R}$ as follows: Upon receiving a transaction, the TEE inserts the transaction into the mempool at a random position. To guarantee that $\mathrm{p_i}$ broadcasts ct_{tx} received from a client, $\mathrm{p_i}$'s TEE mempool $\mathcal{P}_i$ randomly selects 3 nodes for $\mathrm{p_i}$, and only when $\mathcal{P}_i$ verifies the receipt signed by one of them, $\mathcal{P}_i$ accepts the received transaction ct_{tx}.

To enhance the efficiency of cryptographic primitives, we employ a hybrid encryption scheme for implementing the public key encryption scheme. This scheme is utilized for both transaction encryption (line 17) and key distribution (line 48), as depicted in Figure 4. The underlying concept of hybrid encryption is to utilize a public key scheme to share a symmetric key, which is subsequently employed for encrypting the message. Our implementation incorporates AES-GCM [30] for symmetric key encryption and RSA-OAEP for public key encryption. Additionally, we utilize

1. https://anonymous.4open.science/r/SecureMempool-5703.

RSA-PSS in conjunction with SHA-256 for digital signature generation. Also, to ensure robust security, we set the AES key length to 128 bits and the RSA key length to 2048 bits. In our implementation, we employ the Go standard library[2] for cryptographic operations conducted outside mempools. These operations are written in Golang and provide a reliable foundation. For cryptographic operations within mempools, we utilize Mbed TLS[3]. This library, written in C, offers optimized cryptographic functions.

8 Evaluation

We evaluate the performance of our protocol and compare it with Geth. Our evaluation aims to examine the overhead created by running a separate mempool process inside TEEs. We focus on the throughput and latency. Unless otherwise mentioned, the evaluation results in this section are conducted using OpenSGX as the TEE instance.

Experimental settings. We evaluate our protocol using virtual machines (VMs) deployed on Amazon EC2. We use both t2.medium and t3.medium and they have two vCPUs and 4GiB memory. Each VM hosts an Ethereum full node that stores full blockchain data, serving as both a client and a node in our system. In throughput and latency tests, we use 20 full nodes using t2.medium instances to evaluate the protocol performance. Each full node is randomly connected to 3 peers, and every node proposes a block every 5 seconds. In scalability tests, we expand the experiment to include up to 120 full nodes, using a combination of t2.medium and t3.medium instances. In these experiments, every node proposes a block every 20 seconds. Throughout experiments, unless otherwise specified, all transactions have no actual *payload*, allowing us to focus solely on evaluating the protocol's performance in processing transactions.

Throughput. We evaluate the throughput of our protocol and Geth. We vary the number of transactions each client submits to the protocol, denoted as *send rate* in the figures. As shown in Figure 5(d), when the rate is lower than about 30tx/s, the performance of the two systems is almost identical. As the submission rate grows to 46tx/s, both systems reach their peak throughput, where Geth consistently outperforms our protocol. This is expected, as running mempools inside TEEs incur additional costs. We find that the peak throughput of our protocol is about 80% of that in Geth.

Latency. We evaluate the latency of our protocol and Geth. We measure two types of latency: end-to-end latency (measured in μs) and propagation latency (in ms). End-to-end latency is the window from the time a node submits its transaction to the time the transaction enters a node's mempool. The propagation latency, i.e. the time used for transactions/blocks to be broadcast to distributed nodes.

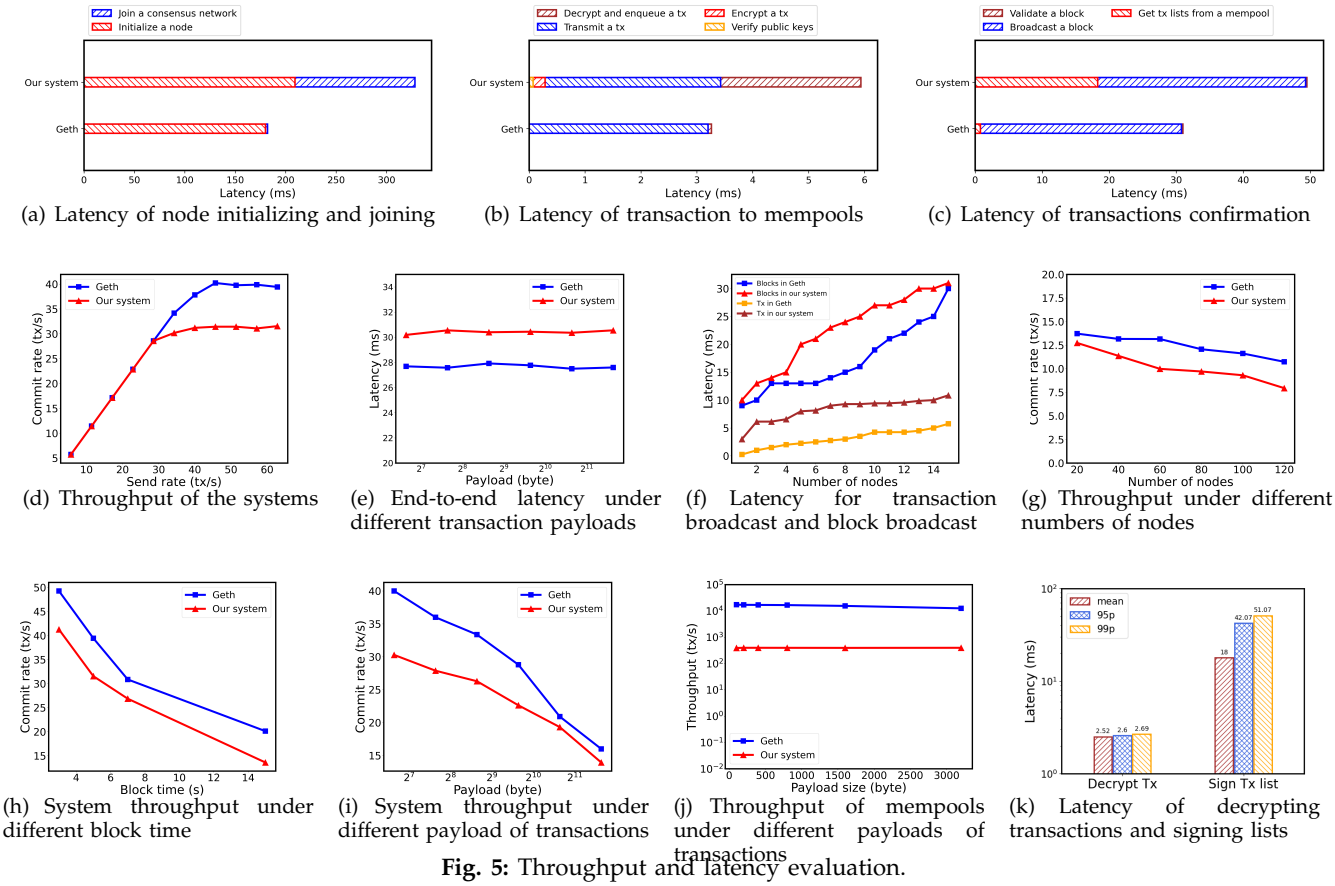
We vary the transactions' size by changing the payload length of each transaction and evaluating performance. As shown in Figure 5(e), the end-to-end latency in Geth and our system remains about 30ms as the payload grows. Our protocol has around 9% higher latency than that for Geth. Figure 5(f) provides an overview of the propagation latency. We conducted an experiment to measure the time required for transactions or blocks to propagate to varying numbers of nodes. The results show a clear trend: as the number of nodes increases, the propagation latency also increases. This outcome is anticipated, as transmitting transactions or blocks to more nodes requires more hops. On average, our protocol exhibits a propagation time approximately 73% longer than that of Geth for transaction propagation and about 25% longer for block propagation, reaching the majority of nodes.

To further understand the overhead of our protocol, we assess the latency breakdown by evaluating the latency in different stages in our protocol. We show the results for *mempool setup*, *transaction submission*, and *transaction confirmation* in Figure 5(a)-5(c), respectively. As shown in Figure 5(a), the time it takes for a node to join the network is noticeably higher (146ms) than that in Geth. This is due to the TEE-based mempool setup, e.g., the mutual attestation process involves additional interactions between different TEEs, and requires cryptographic operations. Fortunately, this step only needs to be conducted once for each node. Additionally, for the latency of the transaction submission and transaction confirmation procedures, the latency of our protocol is marginally higher than that in Geth. The additional overhead caused by our protocol is mainly due to the decryption of transactions and signing ordered transactions inside TEEs (which incurs more than 50% higher latency in our protocol). In contrast, in Geth, no expensive procedures are involved. We observed that the overhead inside TEE may degrade the performance, as shown in Figure 5(j). We further report the latency of decryption and digital signatures in Figure 5(k), which justifies that the overhead of our protocol is mainly created by cryptographic operations.

Performance with different parameters. We vary the intervals of block proposals (denoted as *block time*) and the payloads to assess the performance of the system. As in Figure 5(h) and Figure 5(i), with longer block time or larger transaction payloads, the throughput of both protocols degrades. In our experiments, the performance of our protocol is slightly lower than that of Geth.

Scalability. We also evaluate the scalability of our protocol by varying the number of nodes. We gradually increase the number of nodes by deploying up to 120 full nodes. As shown in Figure 5(g), as the number of nodes grows, the throughput decreases. This is expected as the network bandwidth consumption grows as the number

2. https://github.com/golang/go/tree/master/src/crypto
3. https://github.com/Mbed-TLS/mbedtls



(a) Latency of node initializing and joining (b) Latency of transaction to mempools (c) Latency of transactions confirmation

(d) Throughput of the systems (e) End-to-end latency under different transaction payloads (f) Latency for transaction broadcast and block broadcast (g) Throughput under different numbers of nodes

(h) System throughput under different block time (i) System throughput under different payload of transactions (j) Throughput of mempools under different payloads of transactions (k) Latency of decrypting transactions and signing lists

Fig. 5: Throughput and latency evaluation.

of nodes grows. In all our experiments, the performance of our protocol is only slightly lower than that of Geth. When the number of nodes is 120, the throughput of our protocol reaches 73% of Geth.

Performance with Intel SGX SDK implementation. So far, we have focused on our evaluation using OpenSGX. To have a better understanding of the protocol performance when instantiated on the real SGX platform, we also build a mempool pool using Intel SGX SDK. We compare the performance using Intl SGX implementation with that using OpenSGX. Our results show that the Intel SDK-based transaction pool is 28 times faster than the one on OpenSGX. This implies that in real-world systems, our scheme can achieve greater efficiency. The details are provided in Appendix E.

9 Related Work

Comparison with concurrent studies. MEV services like Flashbots rely on off-chain nodes, known as *relays* to determine the transaction order for Ethereum. Instead of maintaining their own mempool, blockchain nodes rely on relays to maximize their profit. SUAVE [31] and PROF [28] are two early-stage prototypes that use TEE for MEV relays. Specifically, SUAVE uses TEE to ensure that the relays follow the specifications of the platform and behave honestly, including but not limited to obeying the auction rules. PROF utilizes TEE to force miners/relays to follow an arbitrary fair-ordering rule. Our solution also utilizes TEE to order the transactions. However, our work is fundamentally different from SUAVE and PROF in that our work focuses on capturing the concept of transaction fairness. Our TEE-based solution aims to demonstrate the feasibility of achieving our verifiable fairness notion. Also, from an architectural standpoint, our solution possesses the advantage of being adaptable to other blockchain systems beyond Ethereum (as illustrated in Section 5.1), particularly those that do not necessitate the use of MEV relays.

TEE for blockchains. Our discussion develops from the ways of integration between TEE and blockchain.

DApps. Tesseract [32] is a *decentralized application* built on TEEs. Tesseract implements a real-time cryptocurrency exchange that can facilitate secure communication among users while enabling atomic cross-chain transaction orders (namely, all-or-nothing settlement). TEE-backed Tesseract can mitigate the influence of powerful network adversaries who have the capability to eclipse [33] the host. Similarly, the utilization of transparent TEE enclaves can also be used to create new cryptographic functionalities such as sealed-glass proofs (SGPs) [34] and its combination with smart contracts can further build knowledge

marketplaces. Additionally, TEE-based works such as establishing a mixer for Bitcoin have been independently proposed, e.g., Obscuro [35].

Interface. Town Crier (TC) [36] serves as an authenticated data feed for TEE-based smart contracts. TC acts as a trusted bridge between Ethereum and existing HTTPS-enabled data websites, which can securely retrieve data from public *interfaces* and relay concise data information (e.g. prices) to smart contracts. Notably, the inclusion of TEE-protected interfaces and associated communication channels is occasionally incorporated into the system design. The importance of securing these channels has been also acknowledged in [32], [37].

Smart contracts. Ekiden [29] implements a TEE-based platform for private off-chain smart contract executions. It dissects the consensus functionalities and state operations for both high scalability and enhanced privacy (backed by SGX-enabled machines). CONFIDE [37] supports on-chain confidential contract executions. It builds a TEE-based secure data transmission protocol and data encryption protocol to guarantee confidentiality in the transaction life cycle. Besides most efforts dedicated to Ethereum, Fastkitten [38] enables smart contract execution over Bitcoin. TZ4Fabric [39] utilizes ARM Trustzone [14] for the secure execution of smart contracts over Hyperledger Fabric. Secret Network is a TEE-based smart contract platform with active applications [40]. Notably, transactions on Secret Network, such as those for Sienna Swap, have fairness in the sense of being front-running resistant, as they are encrypted.

TABLE 3: TEE adoption in blockchain.

Technical Route (TEE + X)	Usage	Project	Confidentiality	Security	Fairness
DAPP	Exchange and market	[32], [34]	✓	N/A	✗
Interface	Data oracle	[36], [37]	✓	N/A	✗
Smart contract	Confidential states	[29], [37]–[39]	✓	✓	✗
Consensus	Trust, resilience	[41]–[47]	✓	✓	✗
Mempool	Fair DeFi products	[28], [31], Ours	✓	-	✓

Consensus. Using trusted hardware to improve the resilience of Byzantine fault-tolerant (BFT) protocols (i.e., permissioned blockchains) is a conventional topic in the literature [48]–[53]. It was found that instead of requiring $n \geq 3f + 1$, trusted hardware-based BFT only requires $n \geq 2f + 1$, where n is the number of nodes and f is the number of Byzantine failures. This is mainly because the trusted hardware does not *allow* faulty nodes to send inconsistent messages to different nodes. Some of the approaches build the trusted hardware using TEEs [41], [45], [47]. In the permissionless blockchain setting, REM [43] proposes a method to reduce the waste of mining energies in PoW while providing the same security grantees. REM enforces miners to submit trustworthy reports for useful work via the design of hierarchical attestations of TEEs. Hybster [42] presents a hybrid state-machine replication protocol with improved performance that is built on Intel SGX. Meanwhile, TEEs have also been used to improve security and prevent attacks on consensus protocols.

10 Conclusion

In this paper, we present a generic fairness definition, a useful notion for the actual transaction order in blockchains. Our definition weakens previous fairness definitions and leaves the choice of the sequencing rules to the concrete applications. With this definition, one can decouple the order of transactions from the correctness of the blockchain (and consensus protocol). We then present a dedicated protocol that achieves verifiable fairness from trusted hardware. Our evaluation results built on OpenSGX/Intel SGX SDK and Go Ethereum (Geth) show that our protocol only achieves marginal performance degradation on top of Ethereum.

References

[1] Miguel Castro et al. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186. USENIX Association, 1999.

[2] Shehar Bano, Alberto Sonnino, et al. SoK: Consensus in the age of blockchains. In *AFT*, pages 183–198. ACM, 2019.

[3] Mahimna Kelkar, Fan Zhang, et al. Order-fairness for Byzantine consensus. In *CRYPTO*, pages 451–480. Springer, 2020.

[4] Klaus Kursawe. Wendy, the good little fairness widget: Achieving order fairness for blockchains. In *AFT*, pages 25–36. ACM, 2020.

[5] Yunhao Zhang et al. Byzantine ordered consensus without Byzantine oligarchy. In *OSDI*, pages 633–649, 2020.

[6] Mahimna Kelkar et al. Order-fair consensus in the permissionless setting. In *APKC*, volume 2022, pages 3–14. ACM, 2022.

[7] Christian Cachin, Jovana Mi'ci'c, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *FC*. Springer, 2021.

[8] Liyi Zhou et al. High-frequency trading on decentralized on-chain exchanges. In *SP*, pages 428–445. IEEE, 2021.

[9] Philip Daian et al. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *SP*, pages 910–927. IEEE, 2020.

[10] Anton Wahrstätter, Jens Ernstberger, et al. Blockchain censorship. *arXiv preprint arXiv:2305.18545*, 2023.

[11] Miles Carlsten et al. On the instability of Bitcoin without the block reward. In *CCS*. ACM, 2016.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[13] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.

[14] Sandro Pinto et al. Demystifying arm trustzone: A comprehensive survey. In *CSUR*, volume 51. ACM, 2019.

[15] Dayeol Lee et al. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*, 2020.

[16] Prerit Jain et al. OpenSGX: An open platform for SGX research. In *NDSS*, volume 16, pages 21–24. IEEE, 2016.

[17] Christian Cachin et al. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[18] Juan Garay et al. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, pages 281–310. Springer, 2015.

[19] Ben Fisch et al. Iron: Functional encryption using Intel SGX. In *CCS*, pages 765–782. ACM, 2017.

[20] M. K. Reiter and K. P. Birman. How to securely replicate services. In *TOPLAS*, volume 16, pages 986–1009. ACM, 1994.

[21] Christian Cachin et al. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, pages 524–541. Springer, 2001.

[22] Sisi Duan, Michael K Reiter, and Haibin Zhang. Secure causal atomic broadcast, revisited. In *DSN*. IEEE.

[23] Andrew Miller et al. The honey badger of BFT protocols. In *CCS*, pages 31–42. ACM, 2016.

[24] William V Gehrlein. Condorcet's paradox. In *Theory and Decision*, volume 15, pages 161–197. Springer, 1983.

[25] Christian Cachin et al. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Journal of Cryptology*, volume 18, pages 219–246, 2005.

[26] Achour Mostéfaoui et al. Signature-free asynchronous binary byzantine consensus with t¡ n/3, o (n2) messages, and o (1) expected time. In *JACM*, volume 62, pages 1–21, 2015.

[27] Konstantopoulos Georgios and Neuder Mike. Time, slots, and the ordering of events in ethereum proof-of-stake. In *https://www.paradigm.xyz/2023/04/mev-boost-ethereum-consensus*. SIAM, 2023.

[28] PROF: Fair transaction-ordering in a profit-seeking world. *https://initc3org.medium.com/prof-fair-transaction-ordering-in-a-profit-seeking-world-b6dadd71f086*, 2023.

[29] Raymond Cheng, Fan Zhang, et al. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In *EuroSP*, pages 185–200. IEEE, 2019.

[30] An interface and algorithms for authenticated encryption. https://www.rfc-editor.org/rfc/rfc5116, 2008.

[31] The future of MEV is SUAVE. *https://writings.flashbots.net/the-future-of-mev-is-suave/#iii-the-future-of-mev*, 2023.

[32] Iddo Bentov, Yan Ji, et al. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *CCS*. ACM, 2019.

[33] Ethan Heilman, Alison Kendler, et al. Eclipse attacks on Bitcoin's peer-to-peer network. In *USENIX Security*, 2015.

[34] Florian Tramer et al. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *EuroSP*. IEEE, 2017.

[35] Muoi Tran, Loi Luu, et al. Obscuro: A Bitcoin mixer using trusted execution environments. In *ACSAC*, pages 692–701. ACM, 2018.

[36] Fan Zhang, Ethan Cecchetti, et al. Town crier: An authenticated data feed for smart contracts. In *CCS*, pages 270–282. ACM, 2016.

[37] Ying Yan et al. Confidentiality support over financial grade consortium blockchain. In *SIGMOD*. ACM, 2020.

[38] Poulami Das et al. Fastkitten: Practical smart contracts on Bitcoin. In *USENIX Security*, pages 801–818. USENIX Association, 2019.

[39] Christina Müller et al. TZ4Fabric: Executing smart contracts with ARM TrustZone. In *SRDS*, pages 31–40. IEEE, 2020.

[40] Nerla Jean-Louis et al. SGXonerated: Finding (and partially fixing) privacy flaws in TEE-based smart contract platforms without breaking the TEE. *Cryptology ePrint Archive*, 2023.

[41] Jian Liu, Wenting Li, et al. Scalable Byzantine consensus via hardware-assisted secret sharing. In *TC*. IEEE, 2018.

[42] Johannes Behl et al. Hybrids on steroids: SGX-based high performance BFT. In *EuroSys*, pages 222–237. ACM, 2017.

[43] Fan Zhang, Ittay Eyal, et al. REM: Resource-efficient mining for blockchains. In *USENIX Security*, pages 1427–1444, 2017.

[44] Weili Wang et al. Engraft: Enclave-guarded raft on byzantine faulty nodes. In *CCS*, pages 2841–2855. ACM, 2022.

[45] Jérémie Decouchant et al. Damysus: streamlined BFT consensus leveraging trusted components. In *EuroSys*. ACM, 2022.

[46] Huibo Wang et al. Multi-certificate attacks against proof-of-elapsed-time and their countermeasures. In *NDSS*. IEEE, 2022.

[47] Chrysoula Stathakopoulou et al. Adding fairness to order: Preventing front-running attacks in BFT protocols using TEEs. In *SRDS*, pages 34–45. IEEE, 2021.

[48] Miguel Correia et al. How to tolerate half less one Byzantine nodes in practical distributed systems. In *SRDS*. IEEE, 2004.

[49] Byung-Gon Chun et al. Attested append-only memory: making adversaries stick to their word. In *SOSP*. ACM, 2007.

[50] Rüdiger Kapitza et al. CheapBFT: Resource-efficient Byzantine fault tolerance. In *EuroSys*, pages 295–308. ACM, 2012.

[51] Sisi Duan, Karl Levitt, et al. ByzID: Byzantine fault tolerance from intrusion detection. In *SRDS*, pages 253–264. IEEE, 2014.

[52] Dave Levin, John R. Douceur, et al. TrInc: Small trusted hardware for large distributed systems. In *NSDI*, pages 1–14, 2009.

[53] Giuliana Santos Veronese et al. Efficient Byzantine fault tolerance. In *TC*, volume 62, pages 16–30. IEEE, 2013.

[54] Jo Van Bulck et al. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[55] Kit Murdock et al. Plundervolt: Software-based fault injection attacks against intel sgx. In *SP*, pages 1466–1482. IEEE, 2020.

[56] Pietro Borrello et al. {ÆPIC} leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security*, pages 3917–3934, 2022.

[57] Oleksii Oleksenko et al. Varys: Protecting SGX enclaves from practical side-channel attacks. In *ATC*, pages 227–240, 2018.

[58] Rosario Gennaro et al. Secure distributed key generation for discrete-log based cryptosystems. In *EUROCRYPT*, 1999.

[59] AWF Edwards. The meaning of binomial distribution. In *Nature*, volume 186, pages 1074–1074. Springer, 1960.

[60] Lioba Heimbach et al. Eliminating sandwich attacks with the help of game theory. In *AsiaCCS*, pages 153–167. ACM, 2022.

[61] Qin Wang et al. Exploring unfairness on proof of authority: Order manipulation attacks and remedies. In *AsiaCCS*. ACM, 2022.

[62] Xinrui Zhang, Rujia Li, et al. Time-manipulation attack: Breaking fairness against proof of authority Aura. In *WWW*. ACM, 2023.

[63] Noam Nisan, Michael Schapira, and Aviv Zohar. Asynchronous best-reply dynamics. In *WINE*, pages 531–538. Springer, 2008.

A. Verification Violation in Table 1

Our study reveals that the overwhelming majority of in-use systems (87%, 13 out of 15) can not guarantee transaction orders. These systems incorporate ordering algorithms with varying sequencing rules but LACK explicit verification procedures (denoted by ✗). This gap opens the door to potential attacks, including order manipulation and malicious censorship. We analyzed transaction ordering based on the real-world mined block to provide a clear understanding of the negative consequences of missing verification.

In the case of Bitcoin, we randomly selected block #block795221, minted on June 21, 2023, and then examined its actual transaction orders. Our investigation demonstrated that the actual orders, as indicated by their *transaction ID*s, differed from the expected results based on *transaction fee*s. Specifically, the actual transaction order in this block for ID1-ID3 was 3930-7fb8, b722-da4d, and 5c19-e66e, but their fees are 70.1K Sats, 104.3K Sats and 58.2K Sats, respectively. Additionally, similar contradictory results have been observed in other blockchains. For example, we examined #block17525100

in Ethereum and #block2495711 in Litecoin, both of which were random blocks minted on the same day. The evidence supports discrepancies between the predefined rules (e.g., transaction fees) and the actual ordering results within the blocks.

B. Building Blocks

Cryptographic primitives. We present them as follows. *Public key encryption.* A public key encryption scheme PKE includes a tuple of probabilistic polynomial-time (PPT) algorithms $(\mathsf{PKE.Gen}, \mathsf{PKE.Enc}, \mathsf{PKE.Dec})$. $\mathsf{PKE.Gen}$ takes as input a secure parameter λ and outputs a public-private key pair (pk, sk), written as $(pk, sk) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$. $\mathsf{PKE.Enc}$ takes as input a public key pk and a message m and outputs a ciphertext ct, denoted as $ct \leftarrow \mathsf{PKE.Enc}_{pk}(m)$. $\mathsf{PKE.Dec}$ takes as input the private key sk and the ciphertext ct and outputs a message m, written as $m \leftarrow \mathsf{PKE.Dec}_{sk}(ct)$.

Definition 10. *A public key encryption scheme* PKE *is Indistinguishability under adaptive chosen ciphertext attack (IND-CCA2) secure if, for all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that*

$$\Pr[\mathsf{PubK}^{CCA2}_{\mathcal{A},\mathsf{PKE}}(\lambda) = 1] \leq \frac{1}{2} + \mathsf{negl}(\lambda),$$

where $\mathsf{PubK}^{CCA2}_{\mathcal{A},\mathsf{PKE}}(\lambda)$ *is an experiment defined as below:*

- *Run* $\mathsf{S.Gen}(1^\lambda)$ *to obtain a key pair* (pk, sk)*.*
- $\mathcal{A}$ *is given* pk *and access to a decryption oracle* $\mathcal{O}^{\mathsf{PKE.Dec}_{sk}(\cdot)}$*.* $\mathcal{A}$ *outputs a pair of messages* m_0 *and* m_1 *of the same length.*
- *Choose* b *uniformly from* $\{0, 1\}$*, compute* $ct \leftarrow \mathsf{PKE.Enc}_{pk}(m_b)$*, and give* ct *to* $\mathcal{A}$*.*
- $\mathcal{A}$ *can continue to access* $\mathcal{O}^{\mathsf{PKE.Dec}_{sk}}(\cdot)$*, but with a restriction that* ct *cannot be the input of the oracle. Eventually,* $\mathcal{A}$ *outputs a bit* b'*.*
- *The output of the experiment is 1 if satisfying* $b = b'$*, and 0 otherwise. If the output is 1, we say that* $\mathcal{A}$ *succeeds.*

Signature scheme. A signature scheme S consists of three PPT algorithms $(\mathsf{S.Gen}, \mathsf{S.Sign}, \mathsf{S.Verify})$. $\mathsf{S.Gen}$ takes as input λ and outputs a key pair (vk, sk), written as $(vk, sk) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$. $\mathsf{S.Sign}$ takes as input a private key sk and a message m and outputs a signature σ, written as $\sigma \leftarrow \mathsf{S.Sign}_{sk}(m)$. The deterministic algorithm $\mathsf{S.Verify}$ takes as input a verification key vk, a message m, and a signature σ. It outputs 1 if the signature is valid and 0 otherwise, written as $b \leftarrow \mathsf{S.Verify}_{vk}(m, \sigma)$.

Definition 11. *A signature scheme* S *is existential unforgeability under chosen message attack (EUF-CMA) secure if for all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Sig\text{-}forge}^{CMA}_{\mathcal{A},\mathsf{S}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{Sig\text{-}forge}^{CMA}_{\mathcal{A},\mathsf{S}}(\lambda)$ *is an experiment defined as below:*

- *Run* $\mathsf{S.Gen}(1^\lambda)$ *to obtain a key pair* (vk, sk)*.*
- $\mathcal{A}$ *is given* vk *and access to a signature oracle* $\mathcal{O}^{\mathsf{S.Sign}_{sk}}(\cdot)$ *and* $\mathcal{O}^{\mathsf{S.Verify}_{vk}}(\cdot)$*. Then* $\mathcal{A}$ *outputs a legal message* m *with a signature* σ*.*
- *Let* $\mathcal{Q}$ *denote the set of messages queried by* $\mathcal{A}$ *via the oracle* $\mathcal{O}^{\mathsf{S.Sign}_{sk}}(\cdot)$*. The experiment's output is 1 if* $\mathsf{S.Verify}_{vk}(m, \sigma) = 1$ *and* $m \notin \mathcal{Q}$*, and 0 otherwise.*

Trusted hardware properties. A trusted hardware scheme guarantees that loaded programs correctly generate their outputs. Meanwhile, the quotes generated by the scheme should be correctly verified and cannot be forged. Inspired by [19], these properties of a trusted hardware scheme are formally defined as follows:

Definition 12. *A trusted hardware scheme* HW *satisfies the security of execution integrity if for any PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Exec\text{-}integrity}_{\mathcal{A},\mathsf{HW}}(\lambda) = 1] \leq \mathsf{negl}(\lambda),$$

where $\mathsf{Exec\text{-}integrity}_{\mathcal{A},\mathsf{HW}}(\lambda)$ *is an experiment defined as:*

- *Run* $pms \leftarrow \mathsf{HW.Setup}(1^\lambda)$ *and initialize* $\mathcal{O} := \emptyset$*.*
- *Run* $hdl_{\mathsf{prgm}} \leftarrow \mathsf{HW.Init}(pms, \mathsf{prgm})$*.*
- *Run* $output \leftarrow \mathsf{HW.Run}(hdl_{\mathsf{prgm}}, in)$*.*
- $\mathcal{A}$ *is given* hdl_{prgm} *and a valid input* in*. Each time* $\mathcal{A}$ *runs* $\mathsf{HW.Run}(hdl_{\mathsf{prgm}}, in)$ *and gets an* $output$*, and then adds* $output$ *to* $\mathcal{O}$*.*
- *The experiment returns 1 if there exists an* $output'$ *where* $output' \in \mathcal{O}$ *and* $output' \neq output$*, and 0 otherwise.*

Definition 13. *A trusted hardware scheme* HW *achieves remote-attestation-unforgeablility, if for all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that*

$$\Pr[\mathsf{Quote\text{-}forge}_{\mathcal{A},\mathsf{HW}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$$

where $\mathsf{Quote\text{-}forge}_{\mathcal{A},\mathsf{HW}}(\lambda)$ *represents an experiment defined as below:*

- *Run* $pms \leftarrow \mathsf{HW.Setup}(1^\lambda)$ *and initialize* $\mathcal{O} := \emptyset$*.*
- $\mathcal{A}$ *is given* pms*. Each time* $\mathcal{A}$ *queries* $\mathsf{HW.Run\&Quote}$ *and gets a* $quote := (hdl, tag_{\mathsf{prgm}}, in, out, \sigma)$*, add* $(hdl, tag_{\mathsf{prgm}}, in, out)$ *to* $\mathcal{O}$*. Finally,* $\mathcal{A}$ *outputs a* $quote' = (hdl', tag'_{\mathsf{prgm}}, in', out', \sigma')$*.*
- *It outputs 1 if satisfying* $\mathsf{HW.VerifyQuote}(quote') = 1$ *and* $quote' \notin \mathcal{O}$*, and 0 otherwise.*

C. Discussion

The impact of compromised TEE. In our solution, TEE ensures that the user-defined rule $\widetilde{R}$ can be securely applied to any transactions. Essentially, the verifiability of $\widetilde{R}$ is ensured by generating a cryptographic proof using the "quote" by the TEEs. When the TEEs produce ordered results and the corresponding quote is successfully verified, these ordered results are considered fair. However, TEEs suffer from vulnerabilities such as Foreshadow [54], Plundervolt [55], and AEPIC [56]. In our system, we use TEE to build a dedicated protocol to ensure the fairness of transactions. However, even if TEEs are compromised, our solution can fall back to

a plain mode, where the mempool is maintained by each node. In this way, the system fails to achieve IDV-fairness. However, other properties of the blockchain (i.e., consistency and liveness) are not violated.

TEE key management. To ensure the privacy of transactions and the consistency of blockchain services, our system requires all TEE-based mempools to share the decryption key and the signing key via remote attestation. The major drawback is that if one of the TEEs leaks keys (e.g., caused by side-channel attacks [57]), all transactions sent to the nodes are no longer confidential. Below, we discuss alternative key management options.

A threshold cryptosystem with distributed key generation (DKG) [58] is a possible way to mitigate this problem. In a group using a threshold cryptosystem and t-*secure* DKG, the mempools $\{\mathcal{P}_i\}_{i\in[0,\iota]}$ hosted by a group of nodes, generate a public key pk and a set of secret keys $\{sk_i\}_{i\in[0,\iota]}$, where pk is public and sk_i is kept by mempool $\mathcal{P}_i$. Clients can encrypt their transactions with pk and send them to mempools. Then, $\mathcal{P}_i$ decrypts the ciphertext using its secret key sk_i to obtain a share of transactions σ_i. Finally, transactions are recovered by combining $t+1$ shares of transactions from different mempools. The drawback of this approach is that it involves additional communication between TEEs, so the performance may be degraded. Additionally, if nodes leave the system, DKG needs to be executed again, which is expensive, especially when the group size is large.

Another option is that each TEE sets up its key pairs and makes the public keys available to all nodes in the system. When a client needs to communicate with some TEE, the client obtains the public keys of the TEE for corresponding cryptographic operations. The same applies to the communication between TEEs. Such an approach requires the participants to verify public keys of any TEE whenever some interactions are needed.

D. Deterministic Verification Algorithms

Theorem 3. *For any transaction list l with the ordering result r, if a majority of (at least 51%) nodes reach an agreement on the ordering rule $\widetilde{R}$, the verification algorithm outputs a deterministic result with an overwhelming probability.*

We prove Theorem 3. If a majority of (at least 51%) nodes reach an agreement on the ordering rule $\widetilde{R}$, there exists a verification algorithm that outputs a deterministic result. We prove this theorem by contradiction. Namely to prove that if the verification algorithm is probabilistic, there is a negligible probability for the system to achieve the consensus.

Proof. The verification algorithm provides two distinct output scenarios: (i) ***probabilistic*** and (ii) ***deterministic***. In the probabilistic scenario, the algorithm generates results that are uncertain and selected randomly from the set of possible outcomes: {*true*, *false*} on each run. This randomness introduces variability, and the specific outcome cannot be predicted beforehand. On the other hand, in the deterministic scenario, the algorithm consistently produces fixed results that are always either *true* or *false* whenever it is executed. This deterministic behavior ensures a consistent and predictable output every time the algorithm is run.

The probability of scenario-(i) occurring, where the verification algorithm randomly outputs either *true* or *false*, is negligible. In other words, it is highly unlikely that a majority of nodes (51%) would reach a consensus on the sequence based on these random outputs. When querying the verification algorithm, it produces *true* with a 50% probability or *false* with a 50% probability. This behavior follows a binomial distribution [59].

To further analyze this, we can model the number of *true* outputs in a sample of size k drawn from a population of size n. The cumulative distribution function of this distribution can be expressed as follows:

$$\Pr(X \leq k) = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$$

$$\Pr(X > k) = 1 - \Pr(X \leq k).$$

By considering larger values of n, the probability of a majority of nodes (at least 51%n) reaching a consensus based on these random outputs becomes increasingly small, and close to 0, and thus proves Theorem 3. □

E. SGX Performance Comparisons

As our demonstration is done with OpenSGX (an emulation of hardware enclave) and may not fully reflect the real performance, we also build a mempool using Intel SGX SDK and compare its performance with the mempool built with OpenSGX (Table 4). In particular, we keep sending the transactions into two mempools, and evaluate the throughput under different transaction payloads, on an 8-core 1.6GHz local machine with Intel SGX support.

TABLE 4: Throughput comparison of mempools

Payload	OpenSGX (tx/s)	Intel SGX (tx/s)
No payload	2159	61576
200 bytes	1950	58173

Our results reveal that when transactions have no payload, the mempool built with Intel SGX SDK can handle 61576 transactions per second (TPS), while the mempool built with OpenSGX can only handle 2159 TPS. In the case where transactions have a 200-byte payload, the mempool running on real SGX hardware reaches a throughput of 58173 TPS, while the mempool built with OpenSGX has a throughput of 1950 TPS. Overall, SGX SDK-based mempool has a performance 28 times better than the one on OpenSGX-based mempool. This implies that in real-world systems, our solution can achieve greater efficiency and enhanced feasibility.

F. Unfairness Damage

Extracting profits in DeFi products. We use the sandwich attacks in Uniswap to illustrate unfairness damage to decentralized applications. Uniswap is a well-known trading system. It works based on a constant-function automated market maker (AMM) that ensures the balance of trading pairs. The token price in Uniswap is determined by the ratio of tokens in the pool, which is affected by every buying or selling operation.

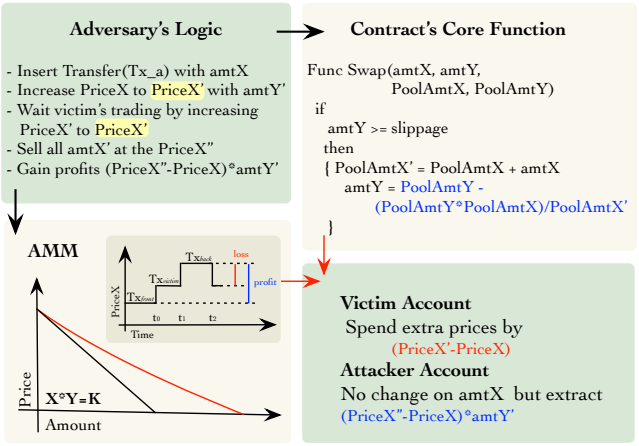


Fig. 6: Sandwich attack in Uniswap

In Uniswap's contact (Figure 6), supposing a client intends to purchase token Y using token X, a set of related variables are: PoolAmtX, PoolAmtY stating the amounts of token pair. Upon detecting this transaction, an adversarial node initiates a transaction T_v to purchase token Y ahead of the client. This results in an artificial increase in demand for token Y, leading the client to receive fewer token Y for the same amount of token X. The price of token Y rises even further, allowing the adversarial node to sell token Y at a profit, acquiring more token X than they spent initially. The adversarial node benefits from sandwiching the client's transaction, manipulating the market to its advantage while causing harm to the client, especially an increase in transaction costs for pair-trading users. We provide an abstract mathematical model for analysis being aligned with [60].

- *Modeling Uniswap's AMM.* We assume that different stakeholders have tokens X and Y (exchanging in pool as X $\rightleftharpoons$ Y), with respective reserves x_t and y_t at time t. The constant value is K where $x_t \cdot y_t = K$. A trader who intends to exchange δ_x tokens of X at time t will receive δ_y tokens of Y. For simplicity, we consider the transaction fees and base fees to be zero. The output of token Y is

$$\delta_y = y_t - \frac{x_t \cdot y_t}{x_t + \delta_x} = \frac{y_t \delta_x}{x_t + \delta_x}.$$

- *Modeling transaction.* A transaction tx to exchange δ_x tokens X entering the mempool at time t_0 is identified as $tx = (x_0, y_0, t_0)$. As time advances, the transactions at time t_1, t_2 are

$$tx^{t_1} = (x_1, y_1, t_1);\ tx^{t_2} = (x_2, y_2, t_2).$$

- *Modeling attack.* In a sandwich attack, a victim normally submits a transaction tx_v expecting to swap δ_y tokens Y at time t_0. The adversary executes the frontrunning transaction tx_a exchanging $\delta^{in}_{a_x}$ token X for δ_{a_y} token Y at time t_1 and a corresponding backrunning transaction at time t_2. As a result, the adversary's profit is the gap between two sandwiching transactions.

$$\Delta_v = \delta^{out}_{a_x} - \delta^{in}_{a_x},\ \text{where}$$

$$\delta^{out}_{a_x} = \frac{x_2 \cdot \delta_{v_y}}{y_2 + \delta_{v_y}} = \frac{\frac{x_0 y_0}{x_0 \delta^{in}_x} \cdot \delta_x}{x_1 + \delta^{in}_{a_x} + \delta_x}.$$

Conversely, the victim will lose

$$\Delta_y = \delta'_{a_y} - \delta_{a_y},\ \text{where}$$

$$\delta_y = y_0 - \frac{x_0 \cdot y_0}{x_0 + \delta_x} = \frac{y_0 \delta_x}{x_0 + \delta_x},$$

$$\delta'_{v_y} = \frac{y_1 \cdot \delta_{v_x}}{x_1 + \delta_{v_x}} = \frac{\frac{x_0 y_0}{x_0 \delta^{in}_x} \cdot \delta_x}{x_1 + \delta^{in}_{a_x} + \delta_x}.$$

Another case is the impact on BRC-20, an experimental fungible token standard for creating new assets on Bitcoin. It embeds JSON data into ordinal inscriptions to enable users to mint, and transfer tokens, and the specific order of transactions has direct financial implications. However, since users input transaction fees based on their preferences, transactions for minting tokens can be front-run. The case will occur when an adversarial user increases their fees to overtake the transaction. Also, if the token is 100% distributed before users complete minting, users will lose the token alongside the gas fees.

Breaking stability. Unfair opportunities cause system instability, especially during consensus procedures [61], [62]. As different nodes form a competitive relationship on the extractable value of transactions, we present our theoretical analyses based on game theory.

The consensus process involves selecting and organizing transactions within a specific time, defined as the block time (b_{exp}). We consider a set of players (i.e., nodes) $\mathcal{N}$ participating in our model. Clients send transactions to the system at a fixed rate of f per second, and only a percentage $p\%$ of these transactions can be extracted, and each extracted transaction yields a value of v. In real-world scenarios, an adversarial node may introduce intentional delays between blocks, represented as Δ. Consequently, the actual block time becomes $b_{real} = b_{exp} + \Delta$. Also, each node has some strategies for node selecting transactions:

- *Strategy* s_0: Nodes collect $f \cdot b_{exp}$ transactions within the expected block time b_{exp} without employing

manipulative tactics. This strategy has an MEV (Miner Extractable Value) of 0.

- *Strategy* s_1: Nodes collect $f \cdot b_{exp}$ transactions within the expected block time b_{exp} while incorporating manipulative tactics to increase profits. This strategy results in an MEV of $f \cdot b_{exp} \cdot p\% \cdot v$.
- *Strategy* s_2: Nodes intentionally delay their mining to the practical block time b_{real}, collecting $f \cdot b_{real}$ transactions. This strategy results in an MEV of $f \cdot b_{real} \cdot p\% \cdot v$.
- *Strategy* s_3: Nodes intentionally delay their mining to their round time, which includes the practical block time b_{real} and the delay Δ. Nodes collect $f \cdot b_{real} \cdot p\%$ transactions, resulting in an MEV of $f \cdot b_{real} \cdot p\% \cdot v$. Here, the node is prone to making a high-quality block by abandoning the non-extractable transactions.

We consider a continuous game where each player, $\mathrm{p_i}$, repeatedly selects a strategy $s_i \in \{s_0, s_1, s_2, s_3\}$ in a a fixed timeframe $\mathcal{D}$. Specifically, during mining time, each node can implement its strategy and inform other nodes of its choice. This process continues with subsequent players, $\mathrm{p_{i+1}}$, $\mathrm{p_{i+2}}$, and so on. We assume the payoff function for each node is $u_i(s_i, s_{-i})$, and s_{-i} denotes the $\mathcal{N}-1$ strategies of all the players except $\mathrm{p_i}$.

In blockchain ecosystems, the value from transaction extraction is fixed in a specific timeframe, denoted as $\mathcal{D}$. As other nodes extract transactions with high rewards, adversarial nodes experience a decrease in their payoffs, prompting them to adapt. Surprisingly, our analysis demonstrates that rational players consistently choose strategy s_3, regardless of whether preceding or subsequent nodes have already extracted transactions. This strategic choice is driven by their pursuit of maximizing their own interests. Consequently, a dominant strategy s_i' emerges, specifically s_3, where $u_i(s_i', s_{-i}) > u_i(s_i, s_{-i})$ holds for all combinations of $s_i \in S_i$ and $s_{-i} \in S_{-i}$. This dominance establishes a stable equilibrium where neither party has the incentive to deviate from this strategy [63].

Under this equilibrium, we examine the throughput of nodes under different scenarios. We refer O_{exp} to the expected throughput when honest nodes operate and O_{real} as the practical throughput when malicious nodes adopt the dominant strategy. Thus, we have

$$O_{exp} = \frac{f * b_{exp}}{b_{exp}} = f \tag{1a}$$

$$O_{real} = \frac{f * b_{real} * p\%}{b_{real}} = f * p\% \tag{1b}$$

Now, we analyze the stability impact when all nodes always select a dominant strategy. We consider two intervals, $[T_i, T_{i+1}]$ and $[T_j, T_{j+1}]$, where $T_{i+1} - T_i = T_{j+1} - T_j = \mathcal{D} \gg b_{real}$. Thus, we have

$$\sum_{i=1}^{\frac{(T_{i+1}-T_i)}{b_{exp}}} i * O_{exp} \geq \sum_{i=1}^{\frac{(T_{j+1}-T_j)}{b_{real}}} i * O_{real}$$

$$\uparrow \qquad\qquad \uparrow$$

$$[T_i, T_{i+1}] \qquad [T_j, T_{j+1}]$$

The presence of throughput fluctuations observed during these two intervals indicates that unfairness undermines the stability of the consensus mechanism. □