

BlindPerm: Efficient MEV Mitigation with an Encrypted Mempool and Permutation

Alireza Kavousi¹, Duc V. Le², Philipp Jovanovic¹, and George Danezis^{1,3}

¹ University College London

² Visa Research*

³ Mysten Labs

Abstract. To mitigate the negative effects of Maximal Extraction Value (MEV), we propose and explore techniques that utilize randomized permutation to shuffle the order of transactions in a committed block before they are executed. We also show that existing MEV mitigation approaches based on encrypted mempools can be extended by permutation-based techniques to provide multi-layer protection. With a focus on BFT style consensus we then propose **BlindPerm**, a framework enhancing an encrypted mempool with permutation at essentially no overheads and present various optimizations. Our protocol neither adds any extra latency nor requires any additional services. Finally, we demonstrate how to extend our mitigation technique to support PoW longest-chain consensus protocols.

1 Introduction

Blockchain and in particular cryptocurrencies initially emerged with a focus on presenting a robust financial ecosystem, but the lack of attention to the issue of *ordering manipulation* turned out to be problematic. It became more serious with the introduction of decentralized finance (DeFi) and in particular decentralized exchange (DEX). The execution mechanism behind these platforms called smart contract [47] has some level of transaction ordering dependence [32] that allows ordering manipulation to cause a major impact on the traded crypto asset by the actors [19]. So, a block proposer can take advantage of such opportunities to achieve some benefit beyond the regular transaction fee and block reward. An illustrative scenario on a DEX known as *sandwich attack* involves front-running (*i.e.*, placing a transaction before) and then back-running (*i.e.*, placing a transaction after) a victim’s transaction to exploit the forced price fluctuations in the traded asset at the cost of victims’ loss. Generally speaking, the profits made via including, excluding, or re-ordering transactions within blocks are described as Maximal Extractable Value (MEV) [19]. Not only block proposers (*i.e.*, miner, validator) can readily extract MEV due to their centralized role in preparing the block, but also any vigilant actor in the system known as *searcher* [1] can

* The main part of the work was conducted while the author was at the University of Bern.

do so with some knowledge of a profitable transaction. This comes from the public nature of the blockchain that offers such knowledge, with speedy connections giving a better advantage. Note that performing the sandwich attack properly requires a correct placement of the transactions with respect to that of victim; otherwise, it may lead to a loss for the attacker. Adoption of services like Flashbots [1] makes the MEV extraction even easier due to the ability to offer front-running as a service [39].

Recent years have seen substantial efforts to combat the negative effects of MEV. Arguably the most notable ones are so-called *order fairness* protocols that consider providing a notion of fairness for the transactions that appear in a finalized block. This notion, however, is not universally agreed upon and may have various interpretations. Timed-order fairness [13, 29, 30] and blind-order fairness [17, 35, 50] are two well-known notions in the literature that aim at providing mitigation at the consensus layer. In essence, the former determines the final ordering of transactions according to their arrival times at the system and the latter hides the content of transactions until their ordering is fixed, preventing any conscious manipulation in the meantime.

Given that the existing proposals following timed-order fairness have expensive configuration costs and usually demand low fault tolerance and high latency, realizing blind ordering in various settings has received plenty of attention [28]. However, there are two principal drawbacks to this approach that might severely affect its usefulness. First, the block proposer can easily front-run other transactions regardless of their contents. Consider a scenario where a popular non-fungible token (NFT) is dropped and the block proposer decides to buy some NFT. They can place their transaction in an early spot, front-running others and buying at a lower price. Second, blinding is concerned with the payload of the transaction, and the leakage of side information (*i.e.*, metadata such as gas price or address) may be enough for the attacker to carry out the attack.

The main intuition behind blind ordering is to make the ordering of transactions independent of their contents. We observe that a random permutation on the committed block also renders the final ordering independent of the committed one. So, we can consider it as a solution at the execution layer where the permutation mitigates any ordering manipulation already occurring in the committed block before it affects the state of the system. This technique is a useful strategy in mitigating the power of the block proposer in imposing their desired ordering. Deploying permutation is also helpful to protect against those types of MEV that negatively affect users, particularly *sandwich* attacks. This is due to the fact that a random permutation can turn a definite profit into a possible loss simply by shuffling the front-running and the back-running transactions, creating a dilemma for the attacker.

We take a step further and argue about the importance of a combined solution, where blind ordering and shuffling enhance each other in a complementary manner. More precisely, an encrypted mempool – that is the core of blind ordering – mitigates content-related attacks such as *spamming*, where an attacker may fill out the mempool in the hope of frontrunning a profitable transaction and thus

diminishing the effectiveness of permutation. Also, permutation on a committed block strengthens the effectiveness of an encrypted mempool by mitigating both issues already mentioned with the blind ordering.

Contributions. The contributions of this work are as follows.

- We introduce randomized permutation as a mechanism for MEV mitigation at the execution layer and propose an efficient construction for BFT style consensus. This technique reduces the power of block proposers in imposing their desired ordering and, in particular, hampers sandwich attacks.
- We introduce **BlindPerm**, a framework that combines encrypted mempool with permutation to provide enhanced MEV mitigation. Particularly noteworthy is the fact that the permutation-based enhancement can come at no additional cost in comparison to regular blind ordering without shuffling.
- We present several optimization techniques that might be of independent interest including *selective encryption* which enables users to only protect their MEV-potential transactions through encryption, thereby offering efficiency gains over a fully encrypted mempool. Also, we show how to extend the permutation-based approach to proof-of-work (PoW) longest-chain protocols.

2 Background

2.1 Threat Model

We consider the setting of a Byzantine fault tolerance (BFT) system. In this setting, there are n parties at most f of which are corrupted by a computationally bounded adversary to do any arbitrary behavior. We assume the existence of authenticated point-to-point channels between each pair of parties. The set of BFT parties is fixed and they are known to each other prior to the protocol execution. The network model is partially synchronous [23], meaning that it may oscillate between periods of synchrony and asynchrony. The common way to treat this is to consider some unknown point as global stabilization time (GST), where it triggers the periods of synchrony that allow message delivery within a known time bound. The optimal fault tolerance in this setting is $f < n/3$ [6].

2.2 Secret Sharing

A (t, n) Shamir secret sharing [44] allows a dealer to distribute a secret s among a set of n shareholders via $\text{SS.Share}(s) \rightarrow s_1, \dots, s_n$, such that it can only be reconstructed uniquely by at least $t + 1$ shares $\text{SS.Combine}(s'_1, \dots, s'_{t+1}) \rightarrow s$, while no information on the secret is revealed otherwise.

Verifiable Secret Sharing (VSS). The basic (t, n) threshold secret sharing scheme of [44] is passively secure, meaning that it works as long as the participating parties run the protocol as specified. In a Byzantine setting participants might be malicious, so the dealer needs to convince parties about the correctness of the sharing and parties need to convince a reconstructor about the correctness

of their released shares. Verifiable secret sharing (VSS) does that by having the dealer commit to the sharing and broadcast it to the parties. Starting from Feldman VSS [24], there have been numerous efforts in the literature to develop VSS schemes with better efficiency in various network models. These are particularly concerned with two aspects of VSS schemes including broadcasting a polynomial commitment to enable share verification and having a complaint phase to deal with any faulty/missing share.

Publicly Verifiable Secret Sharing (PVSS). To extend the scope of verifiability to the public, not only participating parties, PVSS schemes deploy cryptographic primitives such as encryption and non-interactive zero-knowledge proofs (NIZKs). This, in turn, enables anyone to verify the correctness of the distribution phase by the dealer and the reconstruction phase by the set of shareholders. The immediate consequence of public verifiability is getting rid of the complaint phase and therefore reducing latency in VSS schemes at the cost of incurring higher computational overhead to the protocol. A popular PVSS scheme is SCRAPE [14] with the following abstract. To share a random secret s , the dealer runs $\text{PVSS.Share}(s, \{pk_i\}_{i \in [n]})$ and outputs encrypted shares $\{\hat{s}_i\}_{i \in [n]}$ with a proof of correctness π_s . This proof enables anyone to verify the consistency of the shares (*i.e.*, they are evaluations of the same polynomial) and the validity of the ciphertexts (*i.e.*, they contain valid shares). Each shareholder can invoke $\text{PVTSS.Decshare}(\hat{s}_i, sk_i)$ to output a decrypted share \tilde{s}_i with a proof of correctness π_i (*i.e.*, showing correct decryption). Upon gathering $t + 1$ valid decrypted shares, anyone can reconstruct the secret s via $\text{PVTSS.Combine}(\tilde{s}_1, \dots, \tilde{s}_{t+1})$.

2.3 Threshold Cryptography

Distributed Key Generation (DKG). A DKG protocol [38] shares a uniformly distributed secret sk among n parties such that each party receives a partial secret key sk_i , a partial public key pk_i , and a common public key pk while no individual party learns sk . DKGs are commonly used as a trustless setup for threshold encryption and threshold signature schemes.

Threshold Encryption. In a (t, n) threshold encryption scheme TE, one can run the algorithm $\text{TE.Enc}(pk, m) \rightarrow c$ to encrypt a message m under a public key pk resulting from a DKG among a set of n parties. Each party then runs the algorithm $\text{TE.Pardec}(sk_i, c)$ using its own secret key sk_i to obtain a partial decryption pd_i . Finally, c can be decrypted by any threshold set of partial decryptions $\text{TE.Dec}(pd_1, \dots, pd_{t+1}, c)$. Note that it is also possible to verify partial decryption via an additional algorithm $\text{TE.Verify}(pd_i, c)$.

Threshold Signature. In a (t, n) threshold signature scheme TS, any subset of n parties of size $t + 1$ can jointly sign a message m by having each run $\text{TS.Parsign}(sk_i, m)$ to produce a partial signature ps_i , and then $\text{TS.Sign}(ps_1, \dots, ps_{t+1}, m)$ to produce the signature σ . Anyone can verify a partial signature via $\text{TS.Parverify}(pk_i, ps_i, m)$, and the signature via $\text{TS.Verify}(pk, \sigma, m)$.

2.4 Consensus

Consensus is a fundamental problem that aims at providing a set of n parties on possibly different inputs with a common decision despite adversarial behavior by at most f of them. The two core properties of a consensus protocol are *safety* and *liveness*. The former ensures all the honest parties decide on the same value and the latter ensures an honest party eventually decides. Byzantine broadcast is a variant of consensus that enables a sender to send its input to the other parties such that all the honest ones decide on the same value. The FLP impossibility result [25] showed there is no deterministic protocol for Byzantine broadcast in the partial synchrony (and asynchrony) tolerating even one fault. This comes from the fact that in asynchrony it is not possible to distinguish a slow sender from a faulty one, violating the safety in the event of making a decision or liveness in the event of waiting to hear eventually.

State Machine Replication (SMR). Among different formulations of consensus, state machine replication (SMR) [43] enables agreement on *ever-growing* inputs received from external users. It can be naively instantiated by repeated execution of a single-shot consensus protocol, enabling parties to agree on an ordered sequence of inputs proposed by each party. Blockchain protocols implement SMR in two general forms: BFT style such as PBFT [16] or Hotstuff [49], and longest-chain style such as Bitcoin [37] or Ethereum [47]. In the BFT style consensus, which is the focus of this work, BFT parties (*i.e.*, validators) get to agree on a proposal including a batch of transactions. The protocol is typically operated view-by-view and driven by a leader (*i.e.*, block proposer). Although the surge in popularity of blockchain protocols has led to considerable innovations in the literature of SMR [20,45,49], the following protocol flow is a common paradigm in almost all. First, the leader prepares a block of transactions and sends the proposal to all the other parties. Second, each party votes on the proposal if it is properly formed and sends the vote back to the leader. Third, upon collecting $n - f$ votes the leader creates a quorum certificate (QC) and disseminates it to the parties. This process repeats more than once in each view of the protocol to commit. As a concrete instantiation, we deploy the recent work of HotStuff-2 [34] that is an improvement over the original HotStuff [49] following a two-phase commit process per view.

2.5 Memory Pool

In the context of blockchain, a memory pool or mempool refers to where the uncommitted transactions (*i.e.*, pending transactions) exist. The notion of *encrypted mempool* is wildly known as a countermeasure against MEV, addressing the information asymmetry in the blockchain state between the user and the validator via providing privacy for transactions before they are committed [42]. Among different methods in realizing an encrypted mempool, the ones with threshold security are currently offering the most desirable qualities [28]. Low latency, wide coverage, and reasonable performance are the main features that have placed this approach in a promising position.

3 MEV Mitigation with Permutation

A block proposer may take advantage of their full control over transaction ordering when creating a block, which can lead to unfair advantages and unintended profits. For instance, they can simply place their transaction at the top of the block to benefit from a price slippage caused by some large trade on an exchange. When looking closely, however, even leveraging a *perfect* encrypted mempool, *i.e.*, without any metadata leakage, cannot provide enough protection. This is due to the fact that such action does not necessarily rely on the content of existing transactions in the mempool.

A random permutation on the set of transactions in a committed block takes away such forced ordering. Moreover, it hampers sandwich attacks by raising the risk of loss for the attacker. A concrete scenario would be ordering the attacker’s back-run before its front-run, causing a loss to the attacker. In another scenario, the attacker’s front-run might execute but another searcher ends up making a profit by having its back-run execute prior to that of the attacker. The Permutation should occur on the committed block so that the attacker cannot nullify the outcome in case it happens not to be in its favor. It is also crucial to do it *safely* and in a *secure* way. To ensure the former, the honest validators must perform the same permutation on the same block. To ensure the latter, the randomness seeding the permutation should be unpredictable prior to the commit, and unbiased by an adversary controlling the block proposer and possibly a subset of validators. In the following section, we propose a protocol that satisfies these properties.

3.1 Protocol Description

We present our protocol in four main steps as follows. The key point is leveraging the finality of a BFT consensus to have each honest validator safely apply a permutation upon commit with shared randomness computed thereafter.

Step 1 – Submission. Each user broadcasts their transaction tx to the network of $n \geq 3f + 1$ validators.

Step 2 – Committing to the total ordering. The protocol operates in views. Let r be the current view number where a designated validator acts as the leader to propose a block B_r . The block contains a set of transactions in the mempool. Upon receiving the second QC within the view, the proposed block B_r becomes committed by each honest validator.

Step 3 – Deriving the seed. To generate secure shared randomness we assume validators have already run a DKG protocol.⁴ The randomness is securely derived by having validators jointly produce a (t, n) threshold signature on the view number using a unique threshold signature scheme like BLS [9], where $t = 2f$. Thus, when a validator observes that the block B_r has become committed, they

⁴ This is a common assumption made in the state-of-the-art BFT consensus including HotStuff-2 [34].

send a partial signature of the form $H(r)^{sk_i}$ by invoking $\text{TS.Parsign}(sk_i, r)$ to others. Let I be the set of indices of $2f + 1$ valid partial signatures. Anyone can then run $\text{TS.Sign}(\{ps_i\}_{i \in I}, r)$ to produce the signature $\sigma_r = H(r)^{sk}$ using Lagrange interpolation in the exponent. Finally, the seed for permuting block B_r is computed as $\text{seed}_r = H(\sigma_r)$, where $H(\cdot)$ is a cryptographic hash function. A validator enters the next view after deriving the seed.

Step 4 – Execution. Upon computing the seed,⁵ each validator locally performs $\text{Permute}(\text{seed}_r, B_r)$ to randomly shuffle the ordering of transactions in B_r , resulting in a permuted block B'_r . The permuted block is then executed. A standard permutation algorithm is given in Appendix A.

3.2 Analysis

Lemma 1. *The proposed protocol satisfies safety, liveness and a secure permutation.*

Proof. Safety and the liveness of the protocol follow directly from the underlying consensus as we treat it in a black-box manner. Thanks to the finality of the BFT consensus and the uniqueness of the threshold BLS signature, the honest validators apply the same permutation on a committed block they have already agreed on. The seed is pseudorandom and unpredictable to the validator proposing the block before it gets committed, guaranteeing a secure permutation.

4 BlindPerm

A broad scope of MEV comes from the availability of information about transactions, either those that are already submitted on the public mempool or the ones observed early by a capable searcher [10]. Such information could directly affect the users by facilitating the MEV for the validator or searcher through creating dependent transactions or even censoring an undesirable transaction. Consider the scenario where a searcher detects a victim’s transaction and submits a corresponding transaction to the mempool. Any observer may subsequently see this opportunity and do the same. This essentially leads to reducing the effectiveness of the permutation as the chances of the victim’s transaction getting front-runned nevertheless increases.⁶ Since a sole permutation-based solution cannot offer suitable protection in these situations, we propose **BlindPerm**, a framework that enhances an encrypted mempool with permutation. Given that an encrypted mempool may still leak some metadata related to identity or content, this combination is complementary and has the additional benefit of reducing potential negative effects, offering the best of both worlds. From the attacker’s

⁵ A pseudorandom generator (PRG) may apply on the seed first to produce a long random string.

⁶ Any relative ordering of transactions is equally probable and having more dependent transactions from attackers increases the overall chance of frontrunning.

perspective, transaction permutation shifts sandwich attacks from being riskless to being risky, and transaction blinding shifts censorship from being optional to being all-or-nothing.

Our framework includes two categories depending on the way the permutation seed is generated. The first one relies on the validators and the second one relies on the users to contribute to the seed. Interestingly, the latter allows obtaining the seed for free by piggybacking on the encrypted mempool. One important consideration is to ensure a *guaranteed decryption* for each encrypted transaction before a commit by validators; otherwise, it may either lead to an encrypted transaction being buffered indefinitely [35], or the user being able to affect the ordering according to its view of the system [5].

4.1 Seed Contribution by Validators

In this section, we extend the protocol proposed in Section 3.1 to establish an encrypted mempool. We use threshold cryptography to let users encrypt their transactions and validators compute the seed.

Step 1 – Submission. Each user encrypts a transaction tx under the validators’ common public key $\text{TE.Enc}(pk, \text{tx})$ and broadcasts the ciphertext c to the network of $n \geq 3f + 1$ validators.

Step 2 – Committing to the total ordering. The protocol operates in views. Let r be the current view number where a designated validator acts as the leader to propose a block B_r . The block contains a set of encrypted transactions in the mempool. Upon receiving the second QC within the view, the proposed block B_r becomes committed by each honest validator.

Step 3 – Decryption and deriving the seed. When a validator observes that the block B_r has been committed, they produce a decryption share $\text{TE.Pardec}(sk_i, c)$ for each committed tx and a partial signature $\text{TS.Parsign}(sk_i, r)$ as their contribution towards the seed.⁷ They then send the partial decryptions together with partial signatures to others. Each validator can obtain the transaction tx and the seed seed_r by running TE.Dec and TS.Sign upon receiving $2f + 1$ valid partial contributions and enter the next view afterwards.

Step 4 – Execution. Each validator locally performs $\text{Permute}(\text{seed}_r, B_r)$ to randomly shuffle the ordering of transactions in the committed block B_r , resulting in a permuted block B'_r which is then executed.

4.2 Seed Contribution by Users

We now build our BlindPerm protocol with each user choosing a random symmetric-key tx-key to encrypt a transaction and secret share the key to the validators. Our main observation here is to generate the permutation seed as a function of

⁷ For ease of notation, we use the same key-pairs for both threshold encryption and signature. However, they could be different.

the keys tx-key corresponding to the encrypted transactions in the committed block, *e.g.*, XOR of all. This allows computing the seed essentially *at no cost* as the validators no longer produce any threshold signature and use the keys they already retrieved for decryption. As a result, the randomness is uniformly distributed. To implement the secret sharing, we deploy PVSS for concreteness. However, we remark that all the other options presented in Section 7, including secret-sharing with post-verification, can also be used.

Step 1 – Submission. Each user picks a key tx-key to encrypt a transaction tx and broadcasts it to the network of $n \geq 3f + 1$ validators. Moreover, the user runs $\text{PVSS.Share}(\text{tx-key}, \{pk_i\}_{i \in [n]})$ and broadcasts the encrypted shares $\{\hat{s}_i\}_{i \in [n]}$ and proof π_s to the validators.

Step 2 – Committing to the total ordering. The protocol operates in views. Let r be the current view number where a designated validator acts as the leader to propose a block B_r . The block contains a set of encrypted transactions in the mempool whose sharing has been completed at the validators. Upon receiving the second QC within the view, the proposed block B_r becomes committed by each honest validator.

Step 3 – Decryption and deriving the seed. When a validator observes the block B_r has been committed, they produce a decrypted share $\text{PVTSS.Decshare}(\hat{s}_i, sk_i)$ for each committed transaction tx and send it to others. Upon gathering $2f + 1$ valid decrypted shares, the validator obtains tx-key using Lagrange interpolation and decrypts tx . Let $\text{tx-key}_1, \dots, \text{tx-key}_k$ be the set of keys corresponding to the valid transactions in the committed block B_r . Each validator computes the permutation seed as $\text{seed}_r = \text{tx-key}_1 \oplus \dots \oplus \text{tx-key}_k$ and enters the next view thereafter.

Step 4 – Execution. Each validator locally performs $\text{Permute}(\text{seed}_r, B_r)$ to randomly shuffle the ordering of transactions in the committed block B_r , resulting in a permuted block B'_r . The permuted block is then executed.

4.3 Analysis

Lemma 2. *The proposed protocols satisfies safety, liveness and a secure permutation.*

Proof. The safety of the protocols directly follows from that of the underlying consensus. Due to the robustness of the underlying threshold cryptosystems, we are guaranteed to have enough shares for decryption and to derive the seed, ensuring liveness. When users contribute to the seed, they secret share a random key tx-key to the validators. The key is recovered only after committing the block by the validators, guaranteeing the security of the permutation. Moreover, the existence of just one non-colluding user (with validators) enables the seed to be uniformly at random.

5 Optimizations

Selective Encryption. Several works in the literature separate the issue of transaction censorship from the common types of MEV that suffer user experience [28,39,46]. Following this thread we can make some bandwidth optimization in our BlindPerm constructions, particularly the one with users’ contributions towards the seed (Section 4.2). That is, only those users owning an MEV-potential transaction encrypt and let others send their transactions in plaintext. This stems from doing the shuffling after the commit, providing protection against possible front-running, back-running, and sandwich attack against any encrypted transactions. Observe that this does not affect the security of the permutation seed for the following reason. In order for the attacker to make a profit from a victim’s transaction tx (which we assume is encrypted) via the aforementioned strategies, they need to ensure it is indeed *included* in the committed block. This consequently guarantees that the corresponding key tx-key will be considered in the computation of the seed seed , guaranteeing uniform randomness. In fact, even if the validator only includes one encrypted transaction (*i.e.*, victim’s transaction) in the block it is sufficient to ensure the security of the permutation. However, one caveat arises when there is no encrypted transaction included in the committed block. It basically means there is no MEV-potential transaction in the block and thus there is no permutation seed, paving the way for the block proposer to insert their transaction at their desired spot (refer to Section 3).

Timelock Encryption. The concept of timelock encryption or timed encryption [41] allows encrypting a message that is decryptable only after passing some determined time. In other words, it features “encrypting to the future”. To provide a guaranteed delay, traditional schemes rely on sequential computation that is unparallizable. Recently, Gaily et al. [26] presented a construction that offers the same functionality without requiring any sequential computation. In fact, it relies on an existing committee (*i.e.*, threshold network) that produces BLS signatures on time intervals (*i.e.*, discrete view numbers). With the use of an identity-based encryption scheme [8], anyone can encrypt a message to the future under the view number as the identity that can be decrypted only after the release of the corresponding threshold signature as the private key. Given that we already have such threshold network producing BLS signatures in our BlindPerm construction thanks to the validators (Section 4.1), one may leverage it to enable users encrypt their transactions tx to any future view number of their choice. This can be thought of as an on-chain commit-reveal realization for the encrypted mempool [28]. Unlike existing solutions such as [18, 21, 31], this approach does not incur latency or computational overhead to the system. Also, this can pose a considerable boost in communication overhead compared to the typical threshold cryptography paradigm, as the permutation seed and the decryption key for a given view number is only a single BLS signature. By separating the role of validators from threshold network the privacy of transactions last even against a *dishonest majority* of colluding validators. However, an immediate issue that arises with a naive implementation is the possibility of

decrypting a transaction tx at view r without having it included in the committed block by the block proposer, making it vulnerable to MEV extraction afterwards. Moreover, the knowledge of the view number towards which transactions are encrypted may lead to the validator censor the ones encrypted to a certain view number. Fixing these issues without sacrificing the efficiency could be an interesting research question.

Communication-efficient PVSS. SCRAPE [14] is a state-of-the-art PVSS with the following sharing procedure PVSS.Share . The dealer samples a uniform value $s \xleftarrow{\$} \mathbb{Z}_q$, sets the secret as a group element of form $S = h^s$, splits s into shares $\{s_i\}_{i \in [n]}$ using Shamir secret sharing, and computes the encrypted shares under parties' public keys $\{\hat{s}_i = pk_i^{s_i}\}_{i \in [n]}$. The dealer also publishes commitment to shares and $O(n)$ -sized NIZK proofs π_s with individual shares as their witnesses, enabling anyone to check the correctness of sharing with a linear cost. The recent work of [15] introduces efficiency optimizations over SCRAPE to reduce its communication and computation complexities. In particular, the dealer needs to send just $O(1)$ -sized proof of correctness with no public commitments, making the overhead close to optimum [15]. They managed to achieve these efficiencies thanks to making two modifications in the usual model of PVSS, including assigning key pairs to the dealer and doing secret sharing in a group. Fortunately, we can use such PVSS in our BlindPerm construction by accommodating both modifications as the users are equipped with such key-pairs⁸ and symmetric key tx-key to share could be a random group element. We now briefly discuss the high-level idea behind the PVSS proposed in [15]. The authors initially observe that it is possible to check the correctness of sharing in SCRAPE without the involvement of the shareholders' key-pairs. To do so, each encrypted share should be of form $\hat{S}_i = S_i \cdot pk_i^{sk_D}$, establishing a shared Diffie-Hellman key between the dealer and each shareholder to communicate the share. This then turns out to be useful in allowing the dealer to produce one NIZK proof with its secret key sk_D being the witness (instead of individual shares as in SCRAPE) to ensure the correctness of sharing as a whole. We refer the reader to [15] for more details.

6 Discussion

Extension to Longest-chain. Permuting the committed block before execution can also be realized in the longest-chain setting. In a proof-of-work (PoW) blockchain, the miner needs to find a solution (*i.e.*, nonce) to a puzzle to be eligible as the block proposer. Our idea is to use this nonce together with the Merkle root of the transactions (and possibly some auxiliary data) as the seed for the permutation. So, the state change occurs with regard to the permuted block. Should a miner decide to modify the ordering of transactions in the block after learning the seed, they face the threat of loss due to the difficulty rule

⁸ Such key-pairs are nevertheless needed, either ephemeral (for wallets) or registered (for authentication).

of the puzzle. Therefore, this method enhances the recent efforts in leveraging trusted execution environments (TEEs), such as SGX [7], to provide privacy for transactions up to a point where their inclusion in the block is ensured. TEEs can be thought of as a replacement for the committee to generate a key-pair, with the public key being used for encryption.

The above procedure cannot be directly translated to proof-of-stake (PoS) blockchain, as the authentication is no longer based on PoW puzzle. Given that a block proposer is chosen (pseudo) randomly, one may think of utilizing such high entropy randomness as the seed for permutation. But, the leader already learns this which is independent of the block content, paving the way to neutralize the effect of permutation accordingly. Inclusion of the block content (*i.e.*, its Merkle root) in the leader election similar to PoW-based blockchain does not work as the leader can simply go through many candidates to find the best fit *i.e.*, grinding. In a concurrent work, Alpos et al. [3] present a construction that utilizes permutation to prevent sandwich attacks. At a high level, a set of previous leaders contribute towards permutation seed using a commit-reveal mechanism. They use slashing techniques to protect against biasing and splitting the transactions into chunks to increase the permutation space and thus protect against a possible coalition of leaders. Consequently, the system incurs a considerable latency and is also limited in applicability.

Pre-ordered Bundles. Private ordering refer to the process of sending the transactions by the user directly to some trusted services known as relays, like Flashbot [1], rather than publicly broadcasting them to the peer-to-peer network. These services then sequence transactions in bundles and forward them to the validators. Although these approaches can provide MEV protection for users by hiding their transactions from public, their primary goal is to facilitate the MEV extraction by anyone who wishes. That is, the searcher can simply use such services to send a *pre-ordered* bundle of transactions (including its own and that of the victim) to relays without taking the risk of sending over a peer-to-peer network. It is clear that shuffling the order of transactions can be an effective method to get rid of such bundles. On the other hand, treating them as an object (*i.e.*, atomic unit) while shuffling allows appreciating the interior ordering if needed. This might protect the searcher against a front-running attack by the validator [2]. The notion of proposer/builder separation [12] is now wildly adopted on the Ethereum blockchain and aims at reducing the trust on relays by decoupling the role of creating and proposing the blocks by validators.

Limitation and Future Work. Although shuffling is a powerful technique, a block proposer may try to get around it. In an extreme scenario, they can just put one transaction in the block to surely capture an opportunity, say an arbitrage. They can also reduce the effect of permutation by including too many of their own (front-running) transactions in the block. We note in both cases one can argue that they should give up on a large portion of the transaction fees. It is interesting to see how to sidestep such an ability, with an intuitive solution would be making the block preparation decentralized. Our constructions rely on threshold cryptosystems that are difficult to implement in the (PoS)

longest-chain setting with a large population of parties and dynamic participation. Figuring out a design to support this setting is an exciting future work.

7 Related Work

After introducing the MEV problem in [19], a great deal of effort has appeared in the literature to propose countermeasures in various flavors. Here we particularly focus on the solutions at the consensus layer.

Timed-order Fairness. The consensus problem that is at the core of blockchain protocols known as SMR, traditionally does not aim at getting parties agree on a *specific ordering*, but a *total ordering* where all the honest parties are guaranteed to end up with the same sequence of transactions. One way to deal with this is to augment its requirements with an order fairness property. It was shown by two concurrent works of [30, 33] that arguably the most natural definition of fairness known as *receive/relative order fairness* is impossible to achieve. This notion essentially states that for any two transactions tx and tx', if some majority of nodes receive the former sooner than the latter, tx should be ordered before tx'. The impossibility result is due to the so-called Condorcet's cycle/paradox [27], preventing parties to agree on a fair ordering of transactions even when all behave honestly [30].⁹ The impossibility result necessitates the adoption of other variants of timed-order fairness. Kelkar et al. [30] relaxed their definition to capture *batch order fairness* by making "before" to "no later", treating such transactions in batches with relative ordering. In fact, the batch order fairness sidesteps the impossibility result by allowing output transactions in batches and ignoring the possible unfairness resulting from the cycles in each batch. Kelkar et al. [30] introduced Aequitas protocols that order a transaction tx no later than tx' if some fraction γ of parties receive tx before tx', known as γ -batch order fairness. The protocol abstractly operates in three stages. Apart from necessitating a relaxed definition of order fairness, it turns out Condorcet cycles may become larger arbitrarily and also negatively affect the liveness of [30], motivating the design of a follow-up protocol called Themis [29] with a similar spirit. Cachin et al. [13] revisits the notion of order fairness by changing the relative measure of batch order fairness to *differential order fairness*, taking into account the difference between the number of correct parties that receive a tx before tx' compared to that of vice versa. They argue about the usefulness of such modification to tolerate higher fault tolerance compared to that of batch order fairness [29, 30] with a reasonable value for parameter γ , where in their treatment only counts the honest parties.

Blind-order Fairness. The requirement for maintaining causality in SMR systems was first put forth by [40]. They showed the importance of preserving the casual order of users/clients' requests and proposed adding a confidentiality layer to the underlying atomic broadcast (*i.e.*, SMR) to establish a secure causal

⁹ Such a cycle shows up intransitivity in the majoritarian relations, yielding a paradox in selecting a single winner.

atomic broadcast [22]. The recent efforts in literature for blind-order fairness are essentially an extension of this approach, realizing the confidentiality layer with a range of new cryptographic tools and techniques. In [50], validators just carry out the consensus to commit a block of encrypted transactions where a separate secret-management committee runs the decryption per transaction. Such separation could provide optimum fault tolerance of $t < n/2$ for the committee. Fino [35] integrates the blind-order fairness into DAG-based BFT systems that allow parallel dissemination of proposals by multiple validators, achieving high throughput [20, 45]. The proposed blind-order fairness has a hybrid structure, where the key for decryption is either obtained via a fast path using secret-sharing with post-verification or a slow path using threshold decryption. The authors in [36] develop a blind-order fairness system with minimal communication overhead, allowing users to encrypt their transactions to some future time (*i.e.*, view number) with the corresponding private key being released by a committee then. FairPoS [17] introduces a similar notion to blind-order fairness for a longest-chain style consensus called input fairness. They rely on time-based cryptography [11] to hide the content of transactions under a single unknown key until block finalization, which consequently leads to achieving adaptive security. This is implied by the non-parallelizable sequential computation needed for decryption, preventing the leakage of sensitive information (*i.e.*, key material) upon corrupting an honest party. Note that our proposed optimization using timelock encryption share the same rationale with [17, 36] in the sense that a single key (*i.e.*, BLS signature) is enough for decrypting all the encrypted transactions in a committed block.

Threshold Encrypted Mempool. Malkhi and Szalachowski [35] present four approaches to building up an encrypted mempool with threshold security, including threshold cryptography, VSS, secret sharing with post-verification, and Hybrid. We present another approach using PVSS and elaborate on an optimized variant that is relevant for this purpose. In what follows, we briefly describe the two other approaches including secret sharing with post-verification and Hybrid.

Secret Sharing with Post-verification. VSS aims at ensuring the uniqueness, meaning that invoking `SS.Combine` with any threshold number of shares results in the same outcome, and completeness, meaning that any honest party receives a (distinct) valid share from `SS.Share`. The authors in [35] adapt a technique introduced in [48] to relax the requirements and only offer uniqueness. To do so, the dealer runs `SS.Share`, combines all shares in a Merkle tree, certifies the root, and sends with each share a proof of membership, *i.e.*, a Merkle tree path to the root. When a party receives a share, they should verify the Merkle tree proof against the certified root (that is already broadcast by the dealer) before acknowledging it. Moreover, after running `SS.Combine`, each party re-encodes the Merkle tree with the reconstructed secret and compares it with the data sent by the dealer. If the comparison fails, the dealer is faulty. Observe that here the signed Merkle root acts as a public commitment to somewhat relax the use of polynomial commitment. This protocol is the fastest as it uses the efficient and trivial primitives. Note that the sharing completes for each transaction

when there are $n - t$ acknowledgments to ensure $t + 1$ honest validators have received consistent shares, incurring latency. Another issue mentioned in [35] is the possible impact of some specific subset of $t + 1$ validators on the latency of **SS.Combine**. More precisely, since there is no guarantee that all honest validators receive their shares, **SS.Combine** may not be run by the fastest $t + 1$ validators and depend on a specific subset.

Hybrid. In order to address the dependency issue, a hybrid design is proposed where secret sharing with post-verification is augmented with threshold cryptography, enabling any subset of $t + 1$ validators to perform the decryption. Moreover, to maintain safety the protocol requires the results recovered from the **SS.Combine** be equal to **TE.Dec**. To do so, each validator can make use of $t + 1$ secret shares or partial decryptions to check both approaches have the same output. They just need to re-encrypt the key and re-encodes the Merkle tree and check with those originally sent by the dealer.

8 Conclusion

We propose the use of permutation as a mitigation technique at the execution layer against MEV, which is a topic of concern within the Web3 community. We show how blind ordering and shuffling can enhance each other and propose efficient constructions in the BFT style setting. When the users contribute to the seed, our construction allows performing a secure permutation with no further cost than creating an encrypted mempool. Moreover, we present various optimizations and directions for future work.

References

1. Flashbots (2022)
2. Ethereum Bot Gets Attacked for \$20M as Validator Strikes Back. (2023)
3. Alpos, O., Amores-Sesar, I., Cachin, C., Yeo, M.: Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks. arXiv preprint arXiv:2307.02954 (2023)
4. Bacher, A., Bodini, O., Hwang, H.K., Tsai, T.H.: Generating random permutations by coin tossing: Classical algorithms, new analysis, and modern implementation. *ACM Trans. Algorithms* **13**(2), 24–1 (2017)
5. Bebel, J., Ojha, D.: Ferveo: Threshold decryption for mempool privacy in bft networks. *Cryptology ePrint Archive* (2022)
6. Ben-Or, M.: Another advantage of free choice (extended abstract) completely asynchronous agreement protocols. In: *Proceedings of the second annual ACM symposium on Principles of distributed computing*. pp. 27–30 (1983)
7. Bentov, I., Ji, Y., Zhang, F., Breidenbach, L., Daian, P., Juels, A.: Tesseract: Real-time cryptocurrency exchange using trusted hardware. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. pp. 1521–1538 (2019)
8. Boneh, D., Franklin, M.: Identity-based encryption from the weil pairing. In: *Annual international cryptology conference*. pp. 213–229. Springer (2001)

9. Boneh, D., Lynn, B., Shacham, H.: Short signatures from the weil pairing. In: International conference on the theory and application of cryptology and information security. pp. 514–532. Springer (2001)
10. Breidenbach, L., Cachin, C., Chan, B., Coventry, A., Ellis, S., Juels, A., Koushanfar, F., Miller, A., Magauran, B., Moroz, D., et al.: Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. *Chainlink Labs* **1** (2021)
11. Burdges, J., De Feo, L.: Delay encryption. In: Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I. pp. 302–326. Springer (2021)
12. Buterin, V.: Proposer/block builder separation-friendly fee market designs (2021)
13. Cachin, C., Mičić, J., Steinhauer, N., Zanolini, L.: Quick order fairness. In: Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers. pp. 316–333. Springer (2022)
14. Cascudo, I., David, B.: Scrape: Scalable randomness attested by public entities. In: International Conference on Applied Cryptography and Network Security. pp. 537–556. Springer (2017)
15. Cascudo, I., David, B., Garms, L., Konring, A.: Yolo yoso: fast and simple encryption and secret sharing in the yoso model. In: Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part I. pp. 651–680. Springer (2023)
16. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In: *OsDI*. vol. 99, pp. 173–186 (1999)
17. Chiang, J.H.y., David, B., Eyal, I., Gong, T.: Fairpos: Input fairness in proof-of-stake with adaptive security. *Cryptology ePrint Archive* (2022)
18. Cline, D., Dryja, T., Narula, N.: Clockwork: An exchange protocol for proofs of non front-running
19. Daian, P., Goldfeder, S., Kell, T., Li, Y., Zhao, X., Bentov, I., Breidenbach, L., Juels, A.: Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 910–927. IEEE (2020)
20. Danezis, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Narwhal and tusk: a dag-based mempool and efficient bft consensus. In: Proceedings of the Seventeenth European Conference on Computer Systems. pp. 34–50 (2022)
21. Doweck, Y., Eyal, I.: Multi-party timed commitments. *arXiv preprint arXiv:2005.04883* (2020)
22. Duan, S., Reiter, M.K., Zhang, H.: Secure causal atomic broadcast, revisited. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 61–72. IEEE (2017)
23. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* **35**(2), 288–323 (1988)
24. Feldman, P., Micali, S.: Byzantine agreement in constant expected time. In: 26th Annual Symposium on Foundations of Computer Science (sfcs 1985). pp. 267–276. IEEE (1985)
25. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* **32**(2), 374–382 (1985)
26. Gailly, N., Melissaris, K., Romailier, Y.: tlock: practical timelock encryption from threshold bls. *Cryptology ePrint Archive* (2023)
27. Gehrlein, W.V.: Condorcet’s paradox. *Theory and Decision* **15**(2), 161–197 (1983)

28. Heimbach, L., Wattenhofer, R.: Sok: Preventing transaction reordering manipulations in decentralized finance. In: 4th ACM Conference on Advances in Financial Technologies (AFT) (2022)
29. Kelkar, M., Deb, S., Long, S., Juels, A., Kannan, S.: Themis: Fast, strong order-fairness in byzantine consensus. Cryptology ePrint Archive (2021)
30. Kelkar, M., Zhang, F., Goldfeder, S., Juels, A.: Order-fairness for byzantine consensus. In: Advances in Cryptology—CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40. pp. 451–480. Springer (2020)
31. Khalil, R., Gervais, A., Felley, G.: Tex-a securely scalable trustless exchange. Cryptology ePrint Archive (2019)
32. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 839–858. IEEE (2016)
33. Kursawe, K.: Wendy, the good little fairness widget: Achieving order fairness for blockchains. In: Proceedings of the 2nd ACM Conference on Advances in Financial Technologies. pp. 25–36 (2020)
34. Malkhi, D., Nayak, K.: Hotstuff-2: Optimal two-phase responsive bft. Cryptology ePrint Archive (2023)
35. Malkhi, D., Szalachowski, P.: Maximal extractable value (mev) protection on a dag. arXiv preprint arXiv:2208.00940 (2022)
36. Momeni, P., Gorbunov, S., Zhang, B.: Fairblock: Preventing blockchain front-running with minimal overheads. In: International Conference on Security and Privacy in Communication Systems. pp. 250–271. Springer (2022)
37. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Decentralized business review p. 21260 (2008)
38. Pedersen, T.P.: A threshold cryptosystem without a trusted party. In: Advances in Cryptology—EUROCRYPT’91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10. pp. 522–526. Springer (1991)
39. Qin, K., Zhou, L., Gervais, A.: Quantifying blockchain extractable value: How dark is the forest? In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 198–214. IEEE (2022)
40. Reiter, M.K., Birman, K.P.: How to securely replicate services. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(3), 986–1009 (1994)
41. Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)
42. Rondelet, A., Kilbourn, Q.: Threshold encrypted mempools: Limitations and considerations. arXiv preprint arXiv:2307.10878 (2023)
43. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR) **22**(4), 299–319 (1990)
44. Shamir, A.: How to share a secret. Communications of the ACM **22**(11), 612–613 (1979)
45. Spiegelman, A., Giridharan, N., Sonnino, A., Kokoris-Kogias, L.: Bullshark: Dag bft protocols made practical. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 2705–2718 (2022)
46. Wahrstätter, A., Ernstberger, J., Yaish, A., Zhou, L., Qin, K., Tsuchiya, T., Steinhorst, S., Svetinovic, D., Christin, N., Barczentewicz, M., et al.: Blockchain censorship. arXiv preprint arXiv:2305.18545 (2023)
47. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)

48. Yang, L., Park, S.J., Alizadeh, M., Kannan, S., Tse, D.: {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In: 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22). pp. 493–512 (2022)
49. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356 (2019)
50. Zhang, H., Merino, L.H., Estrada-Galinanes, V., Ford, B.: Flash freezing flash boys: Countering blockchain front-running. In: 2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW). pp. 90–95. IEEE (2022)

A Permutation Algorithm

Algorithm 1 Permute [4]

Input: An array a with l elements

Output: A random permutation on the array a

for $i := l$ *downto* 2 *by* -1 **do**

$j := \text{Knuth-Yao}(i) + 1;$

$\text{swap}(a_i, a_j);$

end

Procedure: Knuth-Yao(l)

Input: A positive integer l

Output: Uniform[0, $l - 1$]

$u := 1; x := 0;$

while *true* **do**

while $u < l$ **do**

$u := 2u;$

$x := 2x + \text{randbit};$

end

$d := u - l;$

if $x \geq d$ **then**

return $x - d;$

else

$u := d;$

end

end
