Seahorse: Efficiently Mixing Encrypted and Normal Transactions

Ben Riva[1], Alberto Sonnino[1,2], and Lefteris Kokoris-Kogias[1]

[1] Mysten Labs
[2] University College of London (UCL)

Abstract. Blockchains have been proposed as solution against lack of transparency in the traditional finance domain. However, this does not directly prevent arbitrage, but it at least exposes it publicly. In response MEV (Miner Extractable Value) resilience mechanism have been proposed with one significant class of proposals focusing on encrypting sensitive transactions. These solutions, however, face a critical challenge in balancing transaction privacy, efficiency, and execution speed for non-encrypted transactions. Specifically, prior approaches either compromise privacy for non-committed transactions to achieve low latency or significantly increase communication complexity and processing time to maintain strong privacy guarantees against MEV attacks.
This paper presents a novel hybrid approach specifically designed for MEV-resilience of blockchains. Our method employs a dual encryption scheme for each transaction: a per-transaction encryption that keeps contents private until commitment, and a per-event encryption enabling communication efficient batch processing after commitment. This technique maintains transaction confidentiality from submission until just before execution, while minimizing the delay non-encrypted transactions face. Our construction achieves $O(n + B)$ communication complexity for B encrypted transactions and n nodes in optimistic environments, substantially improving upon existing MEV-resistant protocols.

Keywords: Blockchain · MEV · Consensus · Cryptography

1 Introduction

The primary function of a blockchain involves blockchain nodes receiving transactions from users and proposing them to other nodes, ensuring that all blockchain nodes reach a consensus on an ordered list of *committed* transactions [2]. These committed transactions then update the blockchain's state in a consistent manner based on the established order.

One popular class of applications that use such blockchain infrastructure is Decentralized Finance (DeFi), where apps offering users transparency and the absence of intermediaries have been managing billions of USD. One reason why DeFi is preferred to classic centralized finance (CeFi) is that centralized actors are known to abuse their position either to arbitrage [26] or to censor [18] users.

Unfortunately in DeFi such attacks have also been seen by nodes (and in cases of public mempools, external observers as well) who use their advantage of seeing all transactions before they are committed, and interject other transactions to unfairly gain profit. Those types of attacks are called Maximal Extractable Value (MEV) attacks [14] and include frontrunning and sandwich attacks. A byproduct of MEV attacks is that it incentivizes nodes to slow down block creation to increase their profit.

In this paper, we explore ways to limit MEV. One of the promising approaches for mitigating MEV attacks is to encrypt transactions[3]: The blockchain generates a public key pk under which users can encrypt their transactions. The nodes then commit those encrypted transactions and, once committed, collaborate to decrypt them (i.e., threshold decryption), allowing execution to complete. For simplicity, as we focus on the cryptographic and distributed computation protocols, throughout this work, we assume the extreme case in which encrypted transactions fully hide everything about the transaction except for what is needed for paying basic network fees.

As a public infrastructure, blockchains support a wide variety of transaction types, not all of which require MEV protections (e.g., simple payments or object transfers [8]). However, current MEV protection proposals present a challenge when dealing with mixed transaction lists. For example, consider a sequence of transactions $\mathsf{tx}_1, \mathsf{etx}_2, \mathsf{tx}_3$, where tx_1 and tx_3 are normal transactions and etx_2 is an encrypted one. When this list needs to be finalized for execution, transactions following an encrypted transaction (tx_3) are blocked until the encrypted transaction (etx_2) is decrypted and executed. We define the time between the commitment of tx_3 and its execution as the *delay duration*, which represents the additional latency imposed on normal transactions that follow encrypted ones (assuming execution time is negligible). This delay may be minor for networks with infrequent commits, as the decryption of a block may complete before subsequent commits. However, for blockchains with low latency and high commit rates, tx_3 and consecutive transactions could be significantly delayed by the decryption of etx_2.

There are currently two types of threshold decryption approach used for MEV protection. The first is to use *per-transaction* decryption, where the nodes jointly decrypt each of the encrypted transactions committed separately [6,34,27]. Decrypting a transaction requires $O(1)$ values from each of the nodes. To minimize the delay duration, the nodes can broadcast these values to each other, requiring $O(n)$ communication from each node per transaction, which is a large communication overhead in practice. A cheaper alternative is to send those values to a leader/aggregator, who would then collect messages from all nodes and only broadcast the decrypted transactions. Although this option reduces communication per transaction to $O(1)$ values from each node in the happy case, it might dramatically increase the delay duration in case of a slow or byzantine

[3] Encrypted transactions can help in other use cases such as securely patching a vulnerable smart contract or an insecure onchain state (e.g., [28]), avoiding censorship, etc.

leader/aggregator. Note that the *minimal delay duration* we can hope for is the time it takes to communicate one message between any set of parties (i.e., 1/2 RTT) or even zero in case the threshold decryption and consensus protocols are coupled [6,32]. Hence, the first option above achieves the minimal delay duration but requires large communication, while the second option has a reasonable communication overhead but potentially larger delay duration.

The second type of threshold decryption being used is what we call a *per-event* decryption [29,15] where encryptions are associated with a blockchain event *in the future*, and once that event occurs, the nodes jointly compute an ephemeral key that allows decrypting *all* encryptions associated with that event. Time-lock encryption (e.g., [17]) is an example of time-based events, and in general Identity Based Encryption (IBE) (e.g., [9]) can be used for any type of events defined by the network (e.g., the event that block i was committed can be represented by the unique identity `block:i`). Users would now encrypt their transactions to a close-in-the-future event $\mathcal{E}$ and send them to the blockchain. Some of those transactions would be committed before event $\mathcal{E}$ and be considered valid whereas the rest would be considered invalid. Once $\mathcal{E}$ occurs, the nodes jointly compute the ephemeral key for $\mathcal{E}$, and all encrypted transactions to $\mathcal{E}$ can be decrypted and executed, unblocking also the execution of subsequent committed transactions.

Ideally, to minimize the delay duration, nodes may use a different ordering for execution: All committed transactions that are encrypted for event $\mathcal{E}$ are not scheduled for execution until $\mathcal{E}$ occurs, unblocking the execution of subsequent committed normal transactions. Transactions committed after $\mathcal{E}$ might be delayed due to the decryption process. However, jointly computing the ephemeral key for an event requires only $O(1)$ values from each of the nodes, independent of the number of encrypted transactions that are waiting for that key, thus a simple broadcast (with $O(n)$ communication per node) is sufficient and requires only the minimal delay duration.

The main drawback of per-event decryption is that all encrypted transactions ordered after event $\mathcal{E}$ (or even not committed but still known to nodes) will be considered invalid but still decryptable, since the decryption key for $\mathcal{E}$ is public. As a result, they leak private information about the transaction that could be used to extract MEV (e.g., in case the user just retries the same transaction with a different event $\mathcal{E}'$). Privacy aware users are likely to hesitate to encrypt for a close-by event as they risk not making their transactions on time but losing privacy, considerably increase latency of encrypted transactions. In contrast, per-transaction decryption does not leak such information as only committed transactions are decrypted, but the system needs to either "pay" with large communication overhead or a long delay duration in case of node faults.

In this work we explore a middle ground solution between per-transaction and per-event encryptions, where we minimize the delay duration while guaranteeing privacy for non-committed transactions without introducing chokepoints in the process.

1.1 Our Contribution

We propose a new hybrid approach that achieves low delay duration while maintaining the highest level of privacy and requiring only $O(n+B)$ communication for a batch of B encrypted transactions from each node when the network is synchronous and parties are not faulty. In high-level, each transaction is encrypted twice, requiring both encryptions to be decrypted. The first encryption is using per-transaction key, guaranteeing privacy for non-committed transactions while not affecting the delay duration. The second encryption uses a fast per-event decryption together with other encrypted transactions for the same event. Execution order is defined only with respect to the per-event decryption, thus the delay duration is minimal.

We present a construction that is simple and uses two IBE encryptions, one per-transaction and one per-event. We use the fact that the derived keys of IBE encryption schemes are publicly verifiable, thus decryption can be verified as well given acccess to those decryption keys. The transaction itself is encrypted using symmetric encryption with a fresh key k. We split k into a random string k_1 and a string $k_2 = k \oplus k_1$. The k_1 is encrypted using per-transaction encryption and k_2 using per-event encryption. Once the transaction is committed by the network, nodes start the decryption of k_1 without blocking the execution of normal transactions. This is done efficiently using aggregators needing a small number of communication hops in case of non-faulty parties, and $O(f)$ in the worse case. Once the value of k_1 is committed by the network, the nodes block execution and jointly recover the decryption key associated with the next event, allowing them to decrypt *all* the transactions for that event. Last, the nodes locally decrypt the k_2 component of all relevant transactions, fully decrypt the transactions, and resume execution.
Table 1 compares the main existing approaches with ours.

Scheme	Communication	Privacy	Latency
Per-tx [6,34,27]	$O(nB)$	✓	1 ow
Per-event [15,29]	$O(n)$	✗	1 ow
Our construction	$O(n+B)$ $O(n+Bf)$	✓	2 ow + 1 c $O(cf)$

Table 1: A comparison of the different approaches with minimal duration for n nodes. In all schemes, the additional committed transaction size is $O(1)$. Communication is per node and a batch of B encrypted transactions, ignoring the committed transaction itself. Privacy is of non-committed encrypted transactions. Latency is the additional latency of encrypted transactions on top of the first transaction commit, measured by one-way communication latency (ow) and commit latency (c), assuming that the minimal duration latency is 1 ow. For our construction we compare communication and latency for the happy case and the worst case (in red).

More related work. Other types of protocols that use encrypted transactions include commit-and-reveal protocols that require the user to also decrypt its transactions, resulting in possibly biasable outputs, encryption using Publicly Verifiable Secret Sharing that require large transactions (e.g., [24]), or, depending on third party entities to decrypt transactions (e.g., trusted enclaves, or small MPC committees).

Recently [12] presented a new cryptographic primitive called Batched Threshold Encryption that allows nodes to decrypt a batch of per-event encryptions without revealing information about non-committed encryptions for the same event, using only $O(1)$ elements from each node. This is a very promising direction for achieving our goals, but more work needs to be done on optimizing its performance in practice (e.g., likely more than a minute for a setup with only 50 parties, and more than 18 seconds for a per-event precomputation for the same number of parties).

Last, we mention other mitigations for MEV: MEV-aware application designs (e.g., [19,5,25]), time-based fair ordering of transactions (e.g., [23,11,22]), MEV auctions (e.g. [16,10]), secure enclaves (e.g., [7]). See [4,20,33] for more comprehensive systemization of knowledge of the topic.

Scope limitations. MEV attacks also include scenarios in which nodes can detect public onchain state that can be exploited, and exploit it first as they produce the blocks, e.g., quickly purchase a new collection of NFTs sold on a sale. Those types of attacks do not use user transactions at all, thus the solutions described in this work do not help with them, but they can be combined with other solutions (e.g., randomizing the order after commit [21,27]).

This work assumes deterministic finalization of transactions as privacy relies on the fact that transactions are decrypted only *after* nodes know that all other honest nodes would eventually commit them.

Last, our focus is supporting encrypted transactions in the blockchain protocol layer, as opposed to the application/smart contract layer. The latter may indeed be simpler to design as different applications may process their incoming encrypted requests independently, especially independently of normal transactions. However, it breaks composition since calls to different smart contracts in this case are not atomic.

2 Preliminaries

Bilinear pairings. Let $\mathbb{G}, \mathbb{G}', \mathbb{G}_T$ be groups of prime order q, let $G \in \mathbb{G}$, $G' \in \mathbb{G}', G_T \in \mathbb{G}_T$ be generators and $\mathbb{Z}_q$ be its scalar field. We assume the existence of an efficiently computable bilinear pairing map $e : \mathbb{G} \times \mathbb{G}' \to \mathbb{G}_T$, satisfying (Bilinearity) $\forall (P, Q, a, b) \in (\mathbb{G} \times \mathbb{G}' \times \mathbb{Z}_q \times \mathbb{Z}_q)$: $e(aP, bQ) = b \cdot e(aP, Q) = a \cdot e(P, bQ) = ab \cdot e(P, Q)$, and (Non-degeneracy) $e(G, G') \neq 1$. Last, let $\mathsf{H}_{\mathbb{G}}(\cdot)$ be a hash function that maps strings to $\mathbb{G}$.

For the elliptic curves of BLS12-381, a multiplication in $\mathbb{G}$ or an evaluation of $\mathsf{H}_{\mathbb{G}}$ is roughly twice faster than a multiplication in $\mathbb{G}'$, and about six times faster than an evaluation of the pairing map e.

Threshold IBE Encryption. (t, n)-Threshold Identity Based Encryption (TIBE) is initialized with a public key pk and a set of public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$ such that the i-th party knows sk_i that corresponds to pk_i. Anyone can run $c = \mathsf{Enc}(m, u, \mathsf{pk})$ to encrypt message m using pk for identity u. A party can compute $\mathsf{PartialKey}(\mathsf{sk}_i, u)$ as the i-th partial decryption key k_i^u for identity u. Anyone can verify the partial decryption key k_i^u for identity u using $\mathsf{Verify}(k_i^u, u, \mathsf{pk}_i)$. Given a set D of t valid partial decryption keys, anyone can run $\mathsf{Combine}(D)$ for computing the decryption key k^u for identity u. $\mathsf{Verify}(k^u, u, \mathsf{pk})$ can be used for verifying that key as well. Last, an encryption c for identity u can be decrypted with the decryption key k^u using $\mathsf{Dec}(c, k^u)$.

Recall the IBE encryption scheme of [9]. The secret key is $\mathsf{sk} \in \mathbb{Z}_q$ and the public key is $\mathsf{PK} = \mathsf{sk} \cdot G' \in \mathbb{G}'$. In the threshold setting, sk is distributed between nodes using DKG protocol such that sk_i is a linear t-out-of-n share of sk. An encryption of message m for identity u is

$$\mathsf{Enc}(m, u, \mathsf{PK}) = (u, rG', \mathsf{H}(e(r\mathsf{H}_{\mathbb{G}}(u), \mathsf{PK})) \oplus m)$$

where $r \leftarrow_R \mathbb{Z}_q$ and $\mathsf{H}(\cdot)$ is a hash function with sufficient length, modeled as a random oracle. $\mathsf{PartialKey}(\mathsf{sk}_i, u) = \mathsf{sk}_i \mathsf{H}_{\mathbb{G}}(u)$ and $\mathsf{Verify}(k_i^u, u, \mathsf{PK}_i)$ checks if $e(\mathsf{H}_{\mathbb{G}}(u), \mathsf{PK}_i) = e(\mathsf{PartialKey}(\mathsf{sk}_i, u), G')$. Given the decryption key $k^u = \mathsf{sk} \cdot \mathsf{H}_{\mathbb{G}}(u) \in \mathbb{G}$, $\mathsf{Dec}((u, c_1, c_2), k^u)$ outputs $c_2 \oplus \mathsf{H}(e(k^u, c_1))$.

Blockchain Substrate. We follow the modular view of a blockchain introduced in recent work [1,13], illustrated in Figure 1. Since we do not focus on the transaction dispersal we bundle the three first layers into a Total Order Broadcast (TOB) black-box that continuously accepts transactions from clients and outputs a total order list of blocks of (ordered) transactions. The TOB guarantees that a transaction sent by an honest client is eventually included in a block, and, that all honest nodes eventually output the same list of blocks.

Our contribution is to introduce an intermediate layer between the (TOB) and the the Execution Layer (EL), which we call the MEV Resilient Layer (MEV-R). MEV-R takes as input the total order of blocks and deterministically reorders some of the transactions in a way that mitigates MEV attacks. Our specific focus is on a MEV-R that handles encrypted transactions, but other works have focused on a MEV-R that provides fair timestamping [22]. The final reordered stream of transaction is then provided as input into the EL.

The EL as before simply maintains a state and updates it in a deterministic manner given a continuous, ordered stream of blocks.

We assume that in addition to user transactions, sequenced blocks may include (explicitly or implicitly) unique "synthetic" event messages. An event message indicates an event that is agreed by all honest nodes, e.g., the block height,

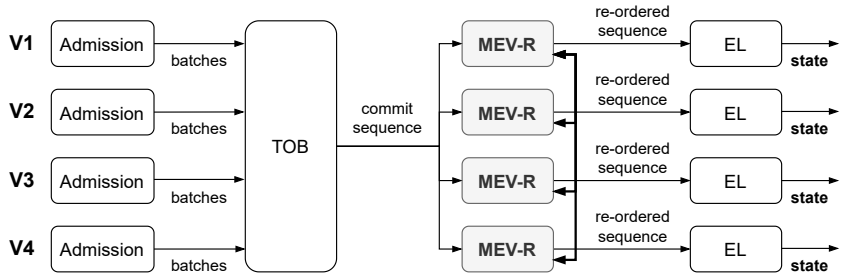

Fig. 1: Overview of a system with 4 nodes (V_1, V_2, V_3, V_4) submitting batches to the Total Order Broadcast (TOB) layer. Each node locally runs a MEV-R layer over the commit sequence and re-order it. Finally the Execution layer (EL) consumes this sequence and outputs the state of the system.

or an approximated time agreed by the chain. For simplicity we assume that events are committed last in their blocks.

Properties of a MEV-R Layer. We informally define the properties of the MEV-R layer:

- **MEV-Resilience:** We say that a protocol has MEV-Resilience if the adversary does not learn information about transactions of honest users that are not yet assigned a sequence number as part of their input in the execution layer. MEV-Resilience has different privacy flavors for transactions that are committed and for transactions failed to commit. For committed valid transactions privacy is only guaranteed until the sequence number is assigned. Afterwards the transaction is revealed so that execution can proceed. For the rest of the transactions, including ones that are committed and ignored by the network (e.g., have an invalid nonce), transactions that are never committed, or even never submitted to the TOB, we want perpetual privacy.[4]
- **Safety:** We say that a protocol is safe if for two honest nodes that get the same ordering of transactions as input from the TOB into the MEV-R layer then they get the same output.
- **Liveness:** We say that a protocol is live if for an honest node that gets an ordering of transactions as input from the TOB into the MEV-R layer, then they will eventually get all input transactions as output.

[4] In the security analysis we use ad-hoc games that are similar to CPA security to capture those informal guarantees. We leave to future work the question of whether our protocols can be proven to be secure for stronger notions of security that use ideal functionalities and simulation. Those notions may, for example, capture "adaptive" attacks in which the adversary chooses which transactions to commit depending on the ciphertexts.

3 Warm Up: Optimizing Execution Order

Most blockchains execute transactions according to a deterministic order that depends on the commit order (e.g., ordered by fee, per block) [31,3,30]. However, when mixing with encrypted transactions, it is better to order the execution of normal transactions first (with the standard ordering logic), and only after that order the encrypted ones. E.g., given a block with transactions $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{etx}_3, \mathsf{tx}_4, \mathsf{etx}_5$, the MEV-R layer would order $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_4$ first, and then $\mathsf{etx}_3, \mathsf{etx}_5$. This change does not break any fairness or security assumptions, while it reduces the delay duration of transactions that are committed in the same block with encrypted ones to zero, as they can be executed independently of the decryption process.

Similarly, when per-event decryption is used, the execution of encrypted transactions should not be ordered after the normal transactions from the same block, but instead at the end of the block that commits the target event. In other words, transactions that are encrypted for event $\mathcal{E}$ and committed before $\mathcal{E}$ is committed, are executed after the normal transactions of block b that includes the target event $\mathcal{E}$. This way, all normal transactions that are committed up to and including block b are not affected by transactions encrypted for $\mathcal{E}$. For example, say we have a block with $\mathsf{etx}_1, \mathsf{tx}_2$, one with $\mathsf{tx}_3, \mathsf{etx}_4, \mathcal{E}$, and another with tx_5. The MEV-R layer will order $\mathsf{tx}_2, \mathsf{tx}_3$ first, $\mathsf{etx}_1, \mathsf{etx}_4$ after, and last tx_5 (which will be blocked on the decryption of the previous two transactions).

Note that even after the above two optimizations, the execution of normal transactions from block b_{i+1} might be blocked on the execution of encrypted transactions from the previous block b_i, impacting the delay duration (as happens with tx_5 in the last example). For blockchains with fast consensus protocols that may commit multiple times per second, this delay can be dramatic. In the next section we show how to reduce this overhead.

4 The Seahorse Design

As the system is designed for blockchains, for simplicity, we assume that all transactions are authenticated. Recall that an encrypted transaction still needs to pay basic network fees even if the encrypted data is invalid. We represent an encrypted transaction etx by $(sender, nonce, blob, sig)$ where *sender* is the entity that sent the transaction and pays the fees, *nonce* is a random nonce chosen by the sender per transaction (which may or may not be used by the network for replay attack prevention), *blob* is the encrypted data, and *sig* is a signature on *sender*, *nonce*, *blob* using the signature key associated in the network with *sender* (*sig* may include other inputs as defined by the network for normal transactions). We denote by $\mathsf{sender}(\cdot)$ the sender of a given transaction and by $\mathsf{nonce}(\cdot)$ the associated nonce.

Let π_i be a deterministic permutation of the node ids, where i is an index, and let $\pi_i[j]$ be the j-th value of π_i. (Those permutations can be constructed, for example, using a pseudorandom function seeded by the hash of the chain at a specific commit, or seeded by i.)

For simplicity and generality, we do not specify the details of the DKG protocol and assume that all parties initialize TIBE and agree on the public key pk. Also, we assume that there is a continuous series of "synthetic" events $\mathcal{E}_1, \mathcal{E}_2, \ldots$ committed by the nodes, e.g. an event for every tenth block. The exact frequency should be set considering the trade-off between the cost per event vs latency.

4.1 Our Construction

While for simplicity the construction below uses TIBE also for per-transaction encryption, any threshold encryption can be used for that layer (e.g., TDH2 from [34]) as long as the encryption can be bounded to the transaction sender and the nonce.

In the following, we explain the protocol in high level. See Figures 2-4 for the algorithms and Figure 5 for an end-to-end example execution that we describe in more detail below.

Encrypting a transaction for event $\mathcal{E}_i$ in the future. Given a transaction tx, the user follows the next steps: First, it selects a random nonce *nonce* of sufficient length (i.e., 40 bits). Second, it selects random strings k_1, k_2 of the length of a symmetric key and sets the identity $u = \mathsf{sender}(\mathsf{tx}) \mid nonce$. Then, the user computes $c_1 = \mathsf{H}(k_1 \mid k_2) \oplus \mathsf{tx}$ where hash function H has a sufficiently long output (and modeled as a random oracle),[5] $c_2 = \mathsf{TIBE.Enc}(k_1, u, \mathsf{pk})$, and $c_3 = \mathsf{TIBE.Enc}(k_2, \mathcal{E}'_i, \mathsf{pk})$ where $\mathcal{E}'_i$ is an event associated with $\mathcal{E}_i$ that will be emitted below, e.g., $\mathcal{E}_i$ can be the message event:i and $\mathcal{E}'_i$ be the message postevent:i. If the length of the transaction tx should remain private as well, we assume that the transaction is padded to a fixed size before being encrypted. Lastly, it outputs $(\mathcal{E}_i, c_1, c_2, c_3)$ as the blob of the encrypted transaction.

```
    // pk is the TIBE's public key.
    // auth is the sender's authentication key.
    // i is the index of a future event.
 1: procedure SubmitTx(auth, pk, tx, i)
 2:     nonce ←R {0,1}^40, k1, k2 ←R {0,1}^128          ▷ Sample random nonce and keys
 3:     Let the identity associated with the i-th event be E'_i = postevent:i.
 4:     u ← sender(tx) | nonce
 5:     c1 ← H(k1 | k2) ⊕ tx
 6:     c2 ← TIBE.Enc(k1, u, pk)                          ▷ Per-transaction encryption
 7:     c3 ← TIBE.Enc(k2, E'_i, pk)                       ▷ Per-event encryption
 8:     blob ← (E_i, c1, c2, c3)
 9:     sig ← Sign((sender(tx), nonce, blob), auth)       ▷ Signature using auth
10:     etx ← (sender(tx), nonce, blob, sig)
11:     SubmitToTOB(etx)
```

Fig. 2: Client protocol.

[5] Other option is to use $\mathsf{H}(k_1 \mid k_2)$ as a key to a symmetric encryption that is used to encrypt tx.

High level flows: For each committed block of transactions, a node calls ProcessNormalTx sequentially for all normal transactions, and then calls ProcessCommit. In parallel it also calls ProcessEtx sequentially for all committed encrypted transactions to initiate the per-transaction decryption. Additionally, the nodes concurrently run AggregatePerTxKey and AggregatePerEventKey to aggregate signatures. Events $\mathcal{E}_1, \mathcal{E}_2, \ldots$ are emitted implicitly by the TOB or the MEV-R layers, while events $\mathcal{E}'_1, \mathcal{E}'_2, \ldots$ are emitted below by the MEV-R layer.

A node maintains locally the following data structures:
- `ongoingEventDec` - an set of indexes of the events for which per-event decryption is running, initially empty.
- $\mathtt{W}$ - a set of transactions for which the per-tx key is not known, initially empty.
- $\mathsf{CT}(\cdot)$ - a set of encrypted transactions for a given event.
- $\mathtt{A}$ - a map between an identity to a set of partial signatures, initially empty.

// Process a normal transaction (step (❸) of Figure 5).
procedure ProcessNormalTx(tx)
  Wait until `ongoingEventDec` is empty
  SubmitToEL(tx) ▷ Output to EL layer

// Process an encrypted transaction (step (❹) of Figure 5).
// This function is also used for rotating aggregators.
// - S - the committed sequence.
// - K - the number of commits before rotating aggregators, protocol configured.
procedure ProcessEtx(S, etx)
  Parse etx as $(sender, nonce, (\mathcal{E}_i, c_1, c_2, c_3), sig)$ ▷ sig is verified by the TOB layer
  AbortTransaction(etx) if $\mathcal{E}_i$ appears before etx in S and **return**
  $m \leftarrow$ CommitIndex(etx) ▷ Index of the committed block containing etx
  $m' \leftarrow$ LastCommitIndex(S) ▷ Index of the last committed block
  require $(m' - m) \mod K == 0$ ▷ Abort call if too early for the next leader
  $u \leftarrow \mathsf{sender}(\mathsf{tx}) \mid nonce$
  $k_j^u \leftarrow \mathsf{TIBE.PartialKey}(\mathsf{sk}_j, u)$
  $ag \leftarrow \pi_m[\frac{m'-m}{K}]$
  Send$(ag, (u, k_j^u))$ ▷ Send the partial decryption to the current aggregator
  $\mathsf{CT}(\mathcal{E}_i) \leftarrow$ etx
  $\mathtt{W} \leftarrow \mathtt{W} \cup \{\text{etx}\}$

// Process committed block.
// - S - the committed sequence.
procedure ProcessCommit(S)
  Wait until `ongoingEventDec` is empty
  for etx $\in \mathtt{W}$ **do**
    if GetPerTxKey(S, etx) returns a value **then** ▷ Per-tx key in the last block
      $\mathtt{W} \leftarrow \mathtt{W} \setminus \{\text{etx}\}$
    else if CommitIndex(etx) $\neq$ LastCommitIndex(S) **then**
      ProcessEtx(S, etx) ▷ Try to rotate aggregator
  for $\mathcal{E}_i \in S$ such that $\mathcal{E}'_i \notin S$ and $\mathsf{CT}(\mathcal{E}_i) \neq \emptyset$ **do** ▷ In order
    Let $C = \mathsf{CT}(\mathcal{E}_i) \cap \mathtt{W}$
    if $C == \emptyset$ **then** ▷ Step (❽) of Figure 5
      Emit($\mathcal{E}'_i$) ▷ Emit the blockchain event $\mathcal{E}'_i =$ postevent:i
      `ongoingEventDec` $\leftarrow$ `ongoingEventDec` $\cup \{i\}$
      $k_j^{\mathcal{E}'_i} \leftarrow \mathsf{TIBE.PartialKey}(\mathsf{sk}_j, \mathcal{E}'_i)$
      Disseminate$(\mathcal{E}_i, k_j^{\mathcal{E}'_i})$ ▷ Send $k_j^{\mathcal{E}'_i}$ to all nodes

Fig. 3: MEV-R Layer of node j with key sk_j, part one.

```
// Called by the designated aggregator (step (❺) of Figure 5).
procedure AggregatePerTxKey((u, k_j^u))
    require TIBE.Verify(k_j^u, u, pk)
    A[u] ← A[u] ∪ {k_j^u}
    return if HasThreshold(A[u]) is false
    k^u ← TIBE.Combine(A[u])
    w ← (u, k^u)
    SubmitToTOB(w)

// Called by all nodes upon receiving E_i, k_j^{E'_i} from node j (step (❾) of Figure 5).
// Incoming messages are collected, but processed only once i ∈ ongoingEventDec.
// - S - the committed sequence.
procedure AggregatePerEventKey(S, E_i, k_j^{E'_i})
    require CT(E_i) ≠ ∅
    require TIBE.Verify(k_j^{E'_i}, E'_i, pk_j)
    A[E'_i] ← A[E'_i] ∪ {k_j^{E'_i}}
    return if HasThreshold(A[E'_i]) is false      ▷ Proceed only if enough partial keys
    k^{E'_i} ← TIBE.Combine(A[E'_i])
    Wait until i = min(ongoingEventDec)           ▷ Guarantee consistent order
    for etx ∈ CT(E_i) do                          ▷ According to a deterministic order
        Parse the blob of etx as (E_i, c_1, c_2, c_3)
        (u, k^u) ← GetPerTxKey(S, etx)
        k_1 ← TIBE.Dec(c_2, k^u)
        k_2 ← TIBE.Dec(c_3, k^{E'_i})
        tx ← H(k_1 | k_2) ⊕ c_1
        SubmitToEL(tx)                            ▷ Output to EL layer
    CT(E_i) ← ∅
    ongoingEventDec ← ongoingEventDec \ {i}
```

Fig. 4: MEV-R Layer of node j with key sk_j, part two.

The blob is cryptographically bound to the transaction sender via the identity u. This binding prevents adversaries from copying transactions with the same identity, as u is computed by the network based on the sender of an authenticated transaction. The security implications of this binding are significant. If u were instead a random string, an adversary could potentially intercept a user's transaction, generate an alternative encrypted transaction with an identical identity, commit it to the network, await its decryption, and subsequently use the revealed decryption key to decrypt the original user's transaction, despite it not being committed. This vulnerability underscores the critical nature of the secure binding between the blob and the transaction sender in maintaining the integrity of the encryption scheme and preventing unauthorized access to transaction contents.

In certain scenarios, the entity encrypting the transaction may differ from the one responsible for network fees. Consequently, setting u to $\mathsf{sender}(\mathsf{etx})$ becomes problematic, as the fee-paying entity could potentially execute the aforementioned attack. To address this issue in such settings, an alternative approach utilizing a signature scheme is proposed. The user encrypting the blob generates a (one-time) signature key pair (sk, pk), employing pk in place of $\mathsf{sender}(\mathsf{etx})$ in

the identity computation. Subsequently, the user signs the encrypted transaction using sk. This method maintains security by preventing adversaries from creating a signed encrypted blob for the same pk, thereby precluding unauthorized use of the network to compute the decryption key for u unless the legitimate user's transaction is committed.

Committing encrypted transactions. Encrypted transactions are committed by the TOB layer as normal ones.

Let the blob of an encrypted transaction be $(\mathcal{E}_i, c_1, c_2, c_3)$. The MEV-R layer aborts the transaction if $\mathcal{E}_i$ was committed already. Otherwise, the MEV-R layer stores internally this encryption and triggers the per-tx decryption below. (Note that MEV-R layer continues processing following transactions.)

Per-tx decryption of etx [slow path]. The MEV-R layer of node j sends $\mathsf{TIBE.PartialKey}(\mathsf{sk}_j, u)$ to node $\pi_m[0]$, where u is calculated as defined above for etx, and m is the index of the commit that included etx. Node $\pi_m[0]$, which we call the *primary* node or aggregator of commit m, checks the partial keys using $\mathsf{TIBE.Verify}$ and once it has enough valid partial decryptions for the encrypted transaction etx, it computes k^u using $\mathsf{TIBE.Combine}$ and sends it to the TOB along with a reference to etx (e.g., its digest). This process is asynchronous and does not block the execution of other transactions.

A malicious node $\pi_m[0]$ may refuse to follow the protocol. To solve this issue, we use the deterministic order of π_m to all other nodes who would act as *backup nodes*. All nodes keep track of all encrypted transactions that are committed and monitor the work of node $\pi_m[0]$. If nodes fail to observe a key k^u associated with one of the encrypted transactions of commit m after a fixed (protocol specified) number of commits, the nodes replace the primary node $\pi_m[0]$ with the backup node $\pi_m[1]$ and send to it the values $\mathsf{TIBE.PartialKey}(\mathsf{sk}_j, u)$ as described above. If needed, this process continues with the next backup node, i.e., if $\pi_m[1]$ fails to publish the expected keys on the TOB within the fixed number of commits, the nodes follow to the next backup node $\pi_m[2]$, and so on. Note that once the decryption keys associated with all encrypted transactions of commit m are committed, the nodes proceed to the next step of the protocol, independently of the party that submitted the decryption keys (thus in case, e.g., the primary node was honest but committed the decryption keys one commit after the deadline, the protocol does not need to wait for the backup nodes).

Inserting k^u into the TOB also allows full nodes and auditors following the output of TOB to ensure that no fully corrupted group of nodes censored encrypted transactions by refusing to decrypt them.

Per-event decryption and execution [fast path]. Let $\mathsf{CT}(\mathcal{E}_i)$ be the set of committed, encrypted transactions for $\mathcal{E}_i$ that were committed up to event $\mathcal{E}_i$. Once the per-tx decryption keys for all the transactions of $\mathsf{CT}(\mathcal{E}_i)$ are committed, and $\mathcal{E}_i$ is also committed (implying no more encrypted transactions for $\mathcal{E}_i$ will be

decrypted), the MEV-R layer implicitly emits $\mathcal{E}'_i$ and pauses executing following transactions. Nodes then jointly generate the per-event decryption key $k^{\mathcal{E}'_i}$. This is achieved by every node disseminates its partial decryption key to all other nodes. When the decryption key is known to the MEV-R layer, it uses it to decrypt pending encrypted transactions for $\mathcal{E}'_i$, outputs their decrypted variants, and resumes to execute subsequent transactions. While the above process is synchronous and blocks the execution of other non-encrypted transactions, it is cheaper than the per-transaction decryption process as it only requires the generation of a single per-event decryption key which can be implemented with a simple broadcast. Note that more than one per-event decryption can be triggered from a single committed block as per-transaction decryption keys from multiple events may be published in any order. The order of events does not impact for the security of the protocol, but for safety all nodes must output the same order of decrypted transactions, and thus nodes keep track of the ongoing per-event decryptions and execute the decrypted transactions according to a deterministic order that depends on the order of committed transactions and their events.

For auditing purposes, nodes may send $k^{\mathcal{E}'_i}$ to the TOB so auditors can verify the decrypted transactions.

Sharding the aggregators. The per-tx decryption process leverages a leader to reduce communication complexity. To avoid placing all the communication and computation burden of collecting and processing partial decryptions over a single node at the time, we may logically create l shards (e.g., $l = 5$). Each shard starts with a different primary node and sequence of backup nodes (i.e., using different permutations $\pi^1_m, \ldots, \pi^l_m$). We then deterministically assign each committed etx to a shard (e.g., in a round-robin fashion). This simple sharding strategy allows a parallel resources utilization of honest nodes and a lower latency.

Putting it all together - an example. Figure 5 illustrates an end-to-end example execution. Say normal (unencrypted) transactions $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3$ and encrypted transactions $\mathsf{etx}_4, \mathsf{etx}_5, \mathsf{etx}_6$ for event $\mathcal{E}$ are sent to the nodes (❶). Nodes call the TOB and receive a list of three blocks, first with $\mathsf{tx}_1, \mathsf{etx}_4$, second with $\mathsf{etx}_5, \mathsf{tx}_2, \mathcal{E}$, and third with $\mathsf{tx}_3, \mathsf{etx}_6$ (❷). When a node calls the MEV-R layer on those blocks, the MEV-R layer outputs $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3$, since those transactions should not be blocked on encrypted transactions for $\mathcal{E}$ (❸). etx_6 is aborted as it was committed after $\mathcal{E}$. Once the MEV-R layer sees $\mathsf{etx}_4, \mathsf{etx}_5$ it sends to nodes $\pi_1[0], \pi_2[0]$ the partial decryptions for those transactions (❹). Assuming those nodes are honest and responsive, they receive enough partial decryptions to reconstruct the full decryption keys w_1, w_2 for the c_2 components of $\mathsf{etx}_4, \mathsf{etx}_5$ and send them to the TOB (❺).

Let the following blocks returned from the TOB include the block tx_7, block w_1, tx_8, w_2, and block tx_9 ($\mathcal{E}'$ is not explicitly included in the second block as it can be deducted from the output of the TOB) (❻). When a node calls the MEV-R layer on those blocks, it outputs $\mathsf{tx}_7, \mathsf{tx}_8$ as those transactions should

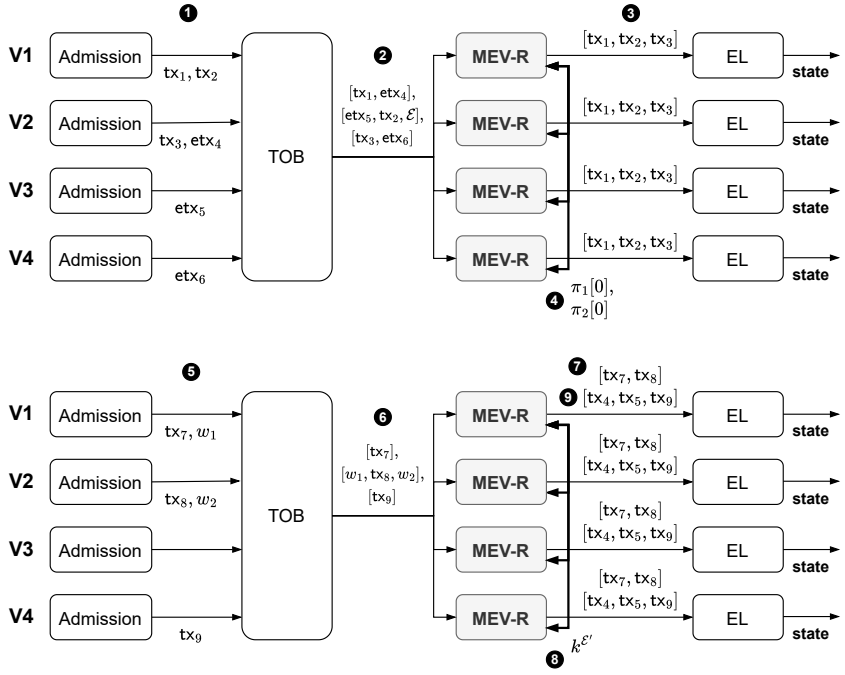

Fig. 5: Example execution performed over the input transactions $\mathsf{tx}_1, \mathsf{tx}_2, \mathsf{tx}_3$ and the encrypted transactions $\mathsf{etx}_4, \mathsf{etx}_5, \mathsf{etx}_6$ for event $\mathcal{E}$, followed by transactions $\mathsf{tx}_7, \mathsf{tx}_8, \mathsf{tx}_9$.

not be blocked on encrypted transactions for $\mathcal{E}$ (❼). Once the MEV-R observes that w_1, w_2 are included in a block (and are valid decryption keys), it pauses executing following transactions (i.e., tx_9 is pending), and broadcasts to all nodes the partial decryption key for event $\mathcal{E}'$ (❽). Once the layer receives enough partial decryption keys to construct the full decryption key for $\mathcal{E}'$, it decrypts $\mathsf{etx}_4, \mathsf{etx}_5$ and gets $\mathsf{tx}_4, \mathsf{tx}_5$, and then outputs transactions $\mathsf{tx}_4, \mathsf{tx}_5, \mathsf{tx}_9$ (❾).

4.2 Analysis

Lemma 1. *The decryption key for identity* $u = \mathsf{sender}(\mathsf{tx}) \mid$ *nonce is revealed only if user* $\mathsf{sender}(\mathsf{tx})$ *sent an encrypted transaction for identity* u *and nonce nonce, and that transaction was committed before its associated event* $\mathcal{E}_i$*.*

This follows the finality and safety of the TOB (i.e., all honest nodes see the same ordered transactions), the fact that transactions are authenticated (i.e., no other party can send a message with the same $\mathsf{sender}(\mathsf{etx})$), the security of TIBE (i.e., less than t parties cannot recover the key for u), and that u is derived explicitly by nodes and therefore cannot be faked by the adversary.

MEV-Resilience of committed transactions. We require that the adversary cannot win the CPA security of IBE with respect to the encryption of k_2 and a

chosen identity $\mathcal{E}^*$: Recall that in that game, the challenger chooses a random bit b, generates a key pair sk, PK and sends PK to the adversary. The adversary can once send (id, m_0, m_1) and ask the challenger for a challenge encryption $\mathsf{Enc}(m_b, id, \mathsf{PK})$. In addition it can repeatedly ask the challenger for a decryption key d^u for any identity u as long as $u \neq id$. The adversary sends b' to the challenger and the challenger outputs $b = b'$.

In our variant, an adversary chooses $\mathcal{E}^*$ as the chosen challenge identity, to represent the fact that $\mathcal{E}'$ was not emitted, auxiliary identity u^* and two messages m_0, m_1 such that $|m_0| = |m_1|$. It gets back the encrypted transaction for those inputs, with $\mathsf{tx} = m_b$. The adversary sends b' to the challenger and the challenger outputs $b = b'$.

Say that adversary $\mathcal{A}$ wins the above game with non-negligible advantage ϵ. We can define a simulator $\mathcal{S}$ that breaks the security of the IBE by emulating the honest parties for $\mathcal{A}$. $\mathcal{S}$ passes the public key from the IBE experiment to $\mathcal{A}$ and tunnels requests to decryption keys directly to the IBE experiment challenger (both modified according to the threshold variant of the protocol). When $\mathcal{A}$ requests a challenge encryption (providing $\mathcal{E}^*, u^*, m_0, m_1$), $\mathcal{S}$ chooses random k_1, k_2, computes $c_2 = \mathsf{Enc}(k_1, u^*, \mathsf{PK})$, sends $(\mathcal{E}^*, k_2, 0)$ (where k_2 and 0 are the two messages for the challenge for identity $\mathcal{E}^*$) to the IBE challenger and receives back an encryption c which is used as c_3, and sets c_1 to $\mathsf{H}(k_1 \mid k_2) \oplus m_b$. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs. When the IBE experiment challenger's bit is 1, c_1 is statistically hiding m_b and thus the adversary cannot guess b with significant advantage. When the bit is 0, the advantage of guessing b is the same as in the above experiment, thus overall the advantage of winning this experiment is $\epsilon/2$. Following the security of the IBE in use, this advantage is negligible.

MEV-Resilience of non-committed transactions. We require that the adversary cannot win an experiment similar to the above one with non-negligible advantage, where the only difference is that here we restrict the requested decryption keys to not include u^* (instead of $\mathcal{E}^*$), to represent the fact that the per-tx decryption key is not revealed if the transaction was not committed before its associated $\mathcal{E}$, following Lemma 1.

We omit the reduction to the security of the IBE encryption as it is similar to the one above.

Safety. Safety directly follows from the safety property of TOB, and the determinism of the MEV-R and EL layers. Let's assume two honest nodes V and V' hold a conflicting state. Honest nodes only execute transactions once they are output by the MEV-R layer, which in turn only processes transactions once they are output from the TOB layer. As a result, there are three cases: (1) V and V' receive different outputs from the TOB layer, which is impossible as it would contradict the safety property of the TOB; (2) The MEV-R layer of V and V' receive the same inputs but produce different outputs. This is impossible following the specification of the protocol, as the MEV-R layer output is deterministic given the inputs from the TOB layer, together with the fact that decryption keys

in use are verifiable, thus the decryption process is deterministic; and, (3) The EL layer of V and V' receive the same inputs from the MEV-R but produce a different state, which would be a violation of the determinism property of the EL layer.

Liveness. Our protocol is live provided that there exists at least t correct nodes. Liveness of the MEV-R layer follows the correctness property of the TIBE scheme (Section 2) under the threshold t and the livenss of the TOB layer. Notice that the TOB layer might reorder inputs or delay them but it will eventually output all of them

The liveness of the TOB layer ensures that an encrypted transaction etx' encrypted for event $\mathcal{E}$ and correctly submitted (possibly multiple times) by a correct user into the TOB is eventually committed before $\mathcal{E}$ is triggered. At this point, all t correct nodes observe it and send their partial decryption key to the primary aggregator. We then distinguish two cases: (i) Assuming the correctness of the TIBE scheme, if the primary aggregator is honest, it decrypts the transaction into a plaintext tx' and submits it to the TOB. (ii) If the primary aggregator is dishonest and drops the transaction, the honest nodes, eventually, send their partial decryption key to some future honest scheduled backup node; we are then back to case (i). Hence eventually all transactios output by TOB will be decrypted.

Since we use broadcast, the per-event decryption eventually progresses given at least t correct nodes.

Communication complexity. A user transaction of length $|\mathsf{tx}|$ bytes requires additional two IBE encryptions of short keys. In case an aggregator is honest, each node sends $O(1)$ group elements per committed encrypted transaction during the per-tx decryption, or $O(f+1)$ group elements in the worst case that f consecutive aggregators are malicious. Last, each node sends $O(n)$ group elements in total per event.

5 Conclusion

Seahorse is a novel hybrid approach tailored for MEV-resilience in blockchains. It integrates a dual-layer encryption scheme per transaction: one layer ensures per-transaction encryption to keep the contents private until commitment, while the other layer applies per-event encryption to enable communication-efficient batch processing post-commitment. This approach preserves transaction confidentiality from submission up to the moment before execution, all while minimizing the delays typically encountered by non-encrypted transactions. Seahorse achieves a communication complexity of $O(n+B)$ for B encrypted transactions and n nodes in optimistic environments, offering a significant improvement over existing MEV-resistant protocols.

References

1. Al-Bassam, M.: Lazyledger: A distributed data availability ledger with client-side smart contracts. arXiv preprint arXiv:1905.09274 (2019)
2. Bano, S., Sonnino, A., Al-Bassam, M., Azouvi, S., McCorry, P., Meiklejohn, S., Danezis, G.: Sok: Consensus in the age of blockchains. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies. pp. 183–198 (2019)
3. Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the libra blockchain. The Libra Assn., Tech. Rep **7** (2019)
4. Baum, C., Chiang, J.H., David, B., Frederiksen, T.K., Gentile, L.: SoK: Mitigation of front-running in decentralized finance. In: Financial Cryptography and Data Security. FC 2022 International Workshops. pp. 250–271. Springer (2022)
5. Baum, C., David, B., Frederiksen, T.K.: P2dex: Privacy-preserving decentralized cryptocurrency exchange. In: ACNS. pp. 163–194. Springer (2021)
6. Bebel, J., Ojha, D.: Ferveo: Threshold decryption for mempool privacy in BFT networks. Cryptology ePrint Archive, Paper 2022/898 (2022), `https://eprint.iacr.org/2022/898`
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. p. 1521–1538. ACM (2019)
8. Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., Williams, B., Zhang, L.: Sui lutris: A blockchain combining broadcast and consensus (2024), `https://arxiv.org/abs/2310.18042`
9. Boneh, D., Franklin, M.: Identity-based encryption from the weil pairing. In: Kilian, J. (ed.) Advances in Cryptology — CRYPTO 2001. pp. 213–229. Springer Berlin Heidelberg (2001)
10. Buterin, V.: Proposer/block builder separation-friendly fee market designs (2021), `https://ethresear.ch/t/proposer-block-builder-separation-friendly-fee-market-designs/9725/1`
11. Cachin, C., Micic, J., Steinhauer, N., Zanolini, L.: Quick order fairness. In: Financial Cryptography and Data Security. pp. 316–333. Springer (2022)
12. Choudhuri, A.R., Garg, S., Piet, J., Policharla, G.V.: Mempool privacy via batched threshold encryption: Attacks and defenses. In: USENIX Security 24. pp. 3513–3529. USENIX Association (2024)
13. Cohen, S., Goren, G., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A.: Proof of availability and retrieval in a modular blockchain architecture. In: International Conference on Financial Cryptography and Data Security. pp. 36–53. Springer (2023)
14. 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)
15. Döttling, N., Hanzlik, L., Magri, B., Wohnig, S.: Mcfly: Verifiable encryption to the future made practical. In: Financial Cryptography and Data Security. pp. 252–269. Springer Nature Switzerland (2024)
16. Flashbots, `https://www.flashbots.net/`

17. Gailly, N., Melissaris, K., Romailler, Y.: tlock: Practical timelock encryption from threshold BLS. Cryptology ePrint Archive, Paper 2023/189 (2023), https://eprint.iacr.org/2023/189
18. Gamestop, amc trading restricted by robinhood, interactive brokers, https://www.investopedia.com/robinhood-latest-broker-to-restrict-trading-of-gamestop-and-others-5100879
19. Heimbach, L., Wattenhofer, R.: Eliminating Sandwich Attacks with the Help of Game Theory. In: ASIA CCS (June 2022)
20. Heimbach, L., Wattenhofer, R.: SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance. In: 4th ACM Conference on Advances in Financial Technologies (AFT), Cambridge, Massachusetts, USA (September 2022)
21. Kavousi, A., Le, D.V., Jovanovic, P., Danezis, G.: BlindPerm: Efficient MEV mitigation with an encrypted mempool and permutation. Cryptology ePrint Archive, Paper 2023/1061 (2023), https://eprint.iacr.org/2023/1061
22. Kelkar, M., Deb, S., Long, S., Juels, A., Kannan, S.: Themis: Fast, strong order-fairness in byzantine consensus. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. p. 475–489. Association for Computing Machinery (2023)
23. Kelkar, M., Zhang, F., Goldfeder, S., Juels, A.: Order-fairness for byzantine consensus. In: Advances in Cryptology – CRYPTO 2020. pp. 451–480. Springer International Publishing (2020)
24. Malkhi, D., Szalachowski, P.: Maximal extractable value (MEV) protection on a DAG. In: 4th International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2022. pp. 6:1–6:17. Schloss Dagstuhl (2022)
25. McMenamin, C., Daza, V., Fitzi, M.: Fairtradex: A decentralised exchange preventing value extraction. ACM CCS Workshop on Decentralized Finance and Security (2022)
26. Payment for order flow — Wikipedia, the free encyclopedia, https://en.wikipedia.org/wiki/Payment_for_order_flow
27. Piet, J., Nair, V., Subramanian, S.: Mevade: An mev-resistant blockchain design. In: 2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–9 (2023)
28. Robinson, D., Konstantopoulos, G.: Ethereum is a dark forest (August 2020), https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest
29. Shutter network: Announcing rolling shutter (2022), https://blog.shutter.network/announcing-rolling-shutter/
30. Sui. https://sui.io (2024)
31. Tikhomirov, S.: Ethereum: state of knowledge and research perspectives. In: Foundations and Practice of Security. pp. 206–221. Springer (2018)
32. Xiang, Z., Das, S., Li, Z., Ma, Z., Spiegelman, A.: The latency price of threshold cryptosystem in blockchains (2024), https://arxiv.org/abs/2407.12172
33. Yang, S., Zhang, F., Huang, K., Chen, X., Yang, Y., Zhu, F.: Sok: Mev countermeasures: Theory and practice (2023), https://arxiv.org/abs/2212.05111
34. Zhang, H., Merino, L.H., Qu, Z., Bastankhah, M., Estrada-Galiñanes, V., Ford, B.: F3b: A low-overhead blockchain architecture with per-transaction front-running protection. In: Conference on Advances in Financial Technologies (2022)