Where to Place Your TEE? In Search of a Censorship-Resilient Design for Rollup Sequencers

Andrei Arusoaie
Alexandru Ioan Cuza University of Iaşi, Romania

Claudiu-Nicu Bărbieru
Alexandru Ioan Cuza University of Iaşi, Romania

Oana-Otilia Captarencu
Alexandru Ioan Cuza University of Iaşi, Romania

Pascal Felber
Université de Neuchâtel, Switzerland

Corentin Libert
UCLouvain, Belgium

Emanuel Onica
Alexandru Ioan Cuza University of Iaşi, Romania

Etienne Rivière
UCLouvain, Belgium

Valerio Schiavoni
Université de Neuchâtel, Switzerland

Peterson Yuhala
Université de Neuchâtel, Switzerland

Abstract

Ethereum is the dominant blockchain ecosystem capable of executing Turing-complete smart contracts. Rollups gained significant traction as the primary layer 2 (L2) solution meant to bring horizontal scalability to the main Ethereum network (L1). A core component of any rollup is the sequencer, which creates new L2 blocks to be submitted in rollup batches to L1. In most of the current rollup architectures, this component is centralised. As a result, these designs are prone to inconspicuous censorship practices by the sequencer. Trusted execution environments (TEEs) can guarantee the integrity of various sequencer components, which is instrumental in addressing censorship. However, the reaction of the system design to censorship attempts depends on where a TEE is integrated and which components it protects. In particular, this reaction is limited in the case of a monolithic TEE-protected sequencer design. Proposer-Builder Separation (PBS) is a non-monolithic paradigm adopted on L1, which separates the production of blocks from proposing them for inclusion in the blockchain. Recently, PBS has been considered for integration with L2 sequencers, with an impact on alleviating censorship. In this paper, we explore the design space of TEE-integrating PBS and non-PBS sequencer variants. First, we introduce a formal framework for the censorship actions that captures the specificity of the L2 sequencer. Then, we analyse to what extent the different designs address these censorship actions. Our main contribution is a novel design variation that allows for a precise observation of censored transactions. In the presence of TEEs, in a PBS setting, we demonstrate this precise observability, which is necessary to enable resilience to censorship.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks; Computing methodologies → Distributed algorithms; Security and privacy → Distributed systems security

Keywords and phrases Rollups, Trusted Execution Environments, Censorship

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2025.27

Funding This work was supported by the CHIST-ERA "SCEAL" project, by a grant from the Romanian Ministry of Research, Innovation and Digitization, CNCS/CCCDI - UEFISCDI, project nr. 86/2025 ERANET-CHISTERA-IV-SCEAL, within PNCDI IV, a grant from the Swiss National Science Foundation under project no. 20CH-1_229451, by the Belgian F.R.S.-FNRS under grant number 40027210, and by the Belgian Wallonia CyberExcellence (grant number 2110186).

1 Introduction

Blockchain networks provide decentralized trust by replicating an immutable ledger of transaction blocks. Transactions sent by users propagate through the network and are kept by nodes in a *mempool*. A consensus algorithm selects nodes to propose new blocks. Blocks are assembled from mempool transactions to maximise producer profit, sometimes through advanced strategies known as maximum extractable value (MEV) [37]. Proposed blocks are then broadcast, validated, and appended to the blockchain.

Ethereum reshaped the blockchain landscape by introducing general-purpose smart contracts. These are small programs, where functions that change the system state are triggered by transactions. This enabled a variety of decentralised applications. The growing popularity of Ethereum caused congestion and rising operational costs. This led to the development of layer 2 (L2) networks designed to enhance scalability [23]. The most prominent L2 solutions are *rollups* [14, 27, 32]. These networks operate as separate blockchains with their own ledgers and state. They provide a decentralised trust anchor by sharing transaction data and committing L2 state roots through contracts on the main Ethereum network (L1).

A central component of rollups is the *sequencer*, responsible for ordering and executing transactions and forming L2 blocks. In most rollups, the sequencer is implemented as a centralised component that maintains a private mempool of transactions. This centralisation provides a good user experience by offering low transaction confirmation latencies at the L2 network level. However, this design is detrimental to the core purpose of provisioning decentralised trust. Centralised sequencers still provide safety guarantees, such as the immutability of the block's content and the legitimacy of state changes. However, centralisation can also impact liveness as it enables sequencers to censor transactions by manipulating their inclusion and ordering in L2 blocks. Other downsides of the centralised sequencer design are its vulnerability as a single point of failure and the concentration of economic power by acting as a single MEV beneficiary in the network.

In this work, we focus on L2 censorship and show that it can take various forms. Censorship can span from the simple exclusion of a transaction from a block to more nuanced variations. The sequencer can take advantage of its private mempool view to perform stealthy reordering of transactions. Within this context, we remark the absence of a formal setting able to capture the possible censorship actions, and in particular the interaction with the mempool.

Trusted Execution Environments (TEEs) are often used to provide integrity guarantees in potentially adversarial settings, making them well-suited to mitigating L2 censorship. TEEs are implemented in processor hardware to offer a secure, isolated area for executing code and loading data into an encrypted memory space at runtime. Integrating TEEs with rollups allows specific components and operations to use authenticated code, ensuring the integrity of produced information and averting censorship actions. However, determining which parts of a rollup should leverage TEEs remains a challenge.

We differentiate between *censorship resistance* and *censorship resilience*, as considered also in the context of L1 by previous work [34]. In the censorship resistance case, the system design denies censorship attempts altogether. In the censorship resilience case, while the system design may allow censorship attempts, it can detect and counter these attempts. Simply running a centralised monolithic sequencer within a TEE primarily ensures censorship resistance. While resistance might seem stronger, it is a more rigid property of a design and can be suppressed by shutting down the TEE by a malevolent owner. Obtaining fine-grained information about censorship attempts enables the system to respond specifically by applying punitive measures proportionate to the offense; for instance, treating an excluded

vote differently from a delayed currency exchange. Such observability fosters censorship resilience in the design and is also influential for a competitive market environment, where rollup implementations may adopt distinct reputation systems. In our work, we show that integrating TEEs at precise places in the architecture facilitates this observability property.

Furthermore, Proposer-Builder Separation (PBS) is one alternative to the monolithic sequencer design, which introduces a new dimension in addressing censorship. Adopted by Ethereum L1 [22], PBS decouples block proposal from construction: distinct *proposer* and *builder* nodes handle these roles, with proposers selecting the most profitable block via *relays* without prior knowledge of included transactions. While the primary purpose of PBS on L1 is decoupling the costly operation of block preparation and creating a competitive block building market [7, 26, 36], it also influences the blockchain's resilience to censorship, by preventing proposers from censoring specific transactions from the mempool. Yet, some authors criticise PBS as mostly shifting the censoring capability to builders and relays [22, 34]. We note that L1 PBS was integrated into an already decentralized setting, with blocks produced by numerous validator nodes [31], far exceeding the number of active builders [22].

L2 rollups provide a centralised setting, with a single sequencer producing each block. Here, PBS expands the builder set, offering the context to increase protection against an untrusted sequencer. While recent L2 PBS works [11, 18, 21] focus on scalability and MEV distribution, anti-censorship has not been systematically studied. Notably, from the perspective of censorship resilience, PBS expands the places in the system where TEEs can be integrated. To date, Rollup Boost [18] is the only L2 PBS in production that uses TEEs.

Our contributions. We introduce a detailed formal model of censorship actions, which, to our knowledge, is the first to capture the mempool interaction specific to the L2 sequencer context. We complement this model with one property, essential for censorship-resilient designs: *transaction censorship observability*, addressed at the granularity of a single transaction. We gradually explore and analyse, in our formal model, several common PBS and non-PBS sequencer variants that employ TEEs to hinder censorship attempts, finding them only censorship-resistant. We propose a novel PBS design that enables censorship-resilience. Our novel design demonstrates that using the TEE exclusively to protect transaction ordering is instrumental to linking censorship actions to the specific blocks in which they occur. We provide a rigorous formal analysis that proves the censorship observability in this design.

Roadmap. §2 provides background on rollups and TEEs. In §3 we describe a censorship threat model. §4 discusses generic PBS and non-PBS sequencer designs. We present our novel design in §5. We compare it with the other designs and discuss its capability to enable censorship resilience in §6. In §7 we cover related work and conclude in §8.

2 Background

Our work is set in the context of Ethereum rollups and leverages the security provided by trusted execution environments (TEEs). In this section, we first elaborate on the generic architecture of a rollup network. Then, we present the principles of TEEs and the guarantees they provide that matter for the design of censorship-resistant and censorship-resilient rollups.

2.1 Ethereum Rollups

Ethereum rollups can be split into two main variants: optimistic and zero-knowledge (zk) based. In both types, transactions are compressed and aggregated into batches. These batches are posted to L1, typically temporarily, persistently storing only their hashes, which

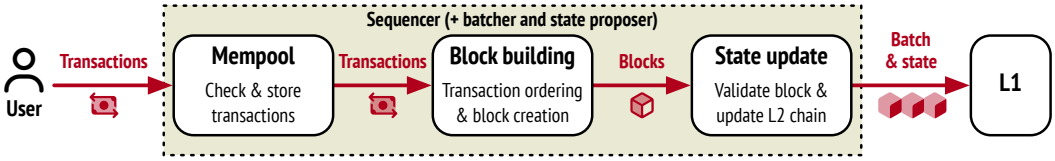

Figure 1 High level overview of a single node rollup sequencer.

can be translated later back to batches by a separate data availability layer. Optimistic rollups consider posted transactions to be valid by default, and provide a challenge time window to prove the contrary. Zk-rollups submit batches along with a zk-proof, which permits verifying batches at posting time, albeit at a higher computational cost. The most significant current rollup solutions according to their use are Arbitrum One, Base Chain, and OP Mainnet, all optimistic rollups [6].

Either type of rollup includes several key components: a *sequencer*, a *batcher*, and a *state proposer*. Users submit transactions to the sequencer, where these are placed in a private mempool. The mempool implementation is usually split into a *pending* transactions subpool of valid transactions ready to be included in blocks, and other subpools that hold invalid transactions. A transaction can be promoted to pending or demoted, at block production interval. The sequencer orders pending transactions, executes these, and forms L2 blocks. The ordering algorithm can vary, e.g., by profit criteria or first-come, first-served. Each rollup also uses different parameters in the block building, such as the maximum block capacity and block frequency. Block capacity is measured in *gas*, the unit for transaction computational and storage costs on Ethereum. Transactions can be added to blocks at most until their total gas cost reaches the capacity limit, at which point the block is considered *full*. The batcher compresses and groups L2 blocks proposed by the sequencer in batches.

The rollup state, usually stored in a database, consists of account-specific values: balances for user accounts and state variables for smart contract accounts. A state root is computed over a Merkle tree representation of the state. The state proposer submits the state root of the rollup chain to L1 using a dedicated smart contract.

The three rollup components are typically run on a single node (with possible replicas for scaling), called a sequencer node [8], since its main function is to operate the sequencer. This operation is usually centralised. Figure 1 presents an overview of this monolithic architecture.

A primary change to this is to keep the proposer role on the sequencer side while delegating block construction to a separate *builder* component. Multiple builders can run on nodes distinct from the sequencer. This proposer–builder separation (PBS), inspired by the L1 approach, is sometimes called sequencer-builder separation in the context of L2 [11].

Many rollups employ escape hatches [20] as a last resort against both sequencer censorship or failure. This mechanism enforces transaction inclusion on L1, but comes with high costs for the user and high confirmation latency. Escape hatches can be the last step of a censorship-resilient solution. However, these are external to the rollup, i.e. provided via smart contracts on L1, and their use also requires knowledge of the censored transaction. In our work, we focus on solutions built within the L2 sequencer context.

2.2 Trusted Execution Environments

A Trusted Execution Environment (TEE) is a hardware-enforced secure code execution context in isolation from the primary operating environment, such as the operating system or hypervisor. This isolation ensures the confidentiality and integrity of both the executing code

and associated data. Most popular TEEs are provided by the major processor vendors. Intel Trust Domain Extensions (TDX) [15] offers isolation at the level of virtual machines (VMs), similar to AMD Secure Encrypted Virtualization-Secure Nested Paging (SEV-SNP) [1]. Intel Software Guard Extensions (SGX) [16] permits separation of trust domains at the application level, running part of its code isolated from the operating system.

Each TEE provides attestation facilities, to guarantee that the code runs in a legitimate TEE and to ensure that it was not tampered with. The identity of an entity that runs the code within a TEE can be bound to the TEE attestation [24].

3 Censorship Threat Model and Context of the Analysis

In this section, we describe in detail the threat model and the analysis context specific to a rollup operation. First, we introduce several notations that we use in the paper. Then, we identify and define a set of possible censorship *actions*. We further define a censorship *observability* property that a rollup design may or may not offer. Afterwards, we identify the possible censorship vectors. We follow by describing several generic facts that apply in our model and can be considered in the analysis of sequencer designs. Finally, we formulate some general security assumptions.

We consider the mempool private to a node, sequencer or builder, which constructs a block. We note that multiple builders are essentially part of a competitive market. While some part of the mempool can be shared, builders can receive transactions via a private order flow, resulting in distinct mempool views.

Notations. B^h denotes the block in the L2 chain at height h. We denote by Mem_P^h the set of pending transactions in a mempool at height h. This set contains valid transactions that are ready to be included in the next block produced by the owner of the mempool. $Mem_P^{h,t}$, designates the mempool at height h and time t, considering the local time of the mempool's owner. Inv^h designates the set of all possible invalid transactions at block height h. Alg_{ord} designates the public ordering algorithm used by either the sequencer or a builder to produce the order of transactions in a block. $Ord(Tx)$ returns the position of a transaction Tx in the ordered list produced by Alg_{ord}. If Tx_1 and Tx_2 are transactions, we write $Tx_1 \prec Tx_2$ to denote that either both transactions are included in the same block B^h and Tx_1 is included before Tx_2, or $Tx_1 \in B^h$, $Tx_2 \in B^{h'}$ and $h < h'$.

Censorship Actions. We now formalise the censorship actions that attackers can perform in our threat model. We note that previous work [34] proposed a coarse split of censorship between obstructing the inclusion of a transaction in a block and delaying its inclusion, formulated within the L1 context. We define the censorship actions in a more expressive form that captures the implications of the mempool view in the L2 context, and addresses situations not covered by previous work.

1. **Simple Censorship (SimpleC)**
   A transaction Tx is subject to simple censorship if, at a block height h, it is pending in the mempool, and is never included in any block produced from h onward, given that it is not deemed invalid and completely discarded at future heights[1]:

   $$Tx \in Mem_P^h \wedge (\nexists h' > h : Tx \in B^{h'} \wedge Tx \in Mem_P^{h'-1})$$

[1] The corner case of not including in Mem_P^h a valid transaction, just received, does not reduce the generality. This is equivalent with adding it and dropping at height h, captured by the definition.

This corresponds to the straightforward case when valid transactions are dropped intentionally. This does not imply any deviations from Alg_{ord}.

2. **Ordering Censorship (OrderC)**
We consider two transactions $Tx_1, Tx_2 \in Mem_P^h$ and a known ordering algorithm, Alg_{ord}, which decides that Tx_1 is in front of Tx_2 ($Ord(Tx_1) < Ord(Tx_2)$). Transaction Tx_1 is affected by ordering censorship if Tx_1 is included after Tx_2 (in the same block with Tx_2 or in a block produced afterwards):

$$Tx_1, Tx_2 \in Mem_P^h \wedge Ord(Tx_1) < Ord(Tx_2) \wedge Tx_2 \prec Tx_1$$

This corresponds to the situation when a transaction is intentionally delayed for censorship purposes by deviating from the correct implementation of Alg_{ord}. In this case the attacker does not modify the transaction set reaching the mempool. A possible example is observing a voting process and delaying one transaction through reordering, even by a single position, e.g., after another transaction that corresponds to closing the polls; practically, this means nullifying the delayed transaction.

3. **Frontrunning Censorship (FrontC)**
We consider a party M interested in delaying a transaction Tx_1 by injecting a transaction Tx_2. We denote by $Inject(M)$ the set of possible valid transactions that are injected by the mempool observer M. Transaction Tx_1 is subject of frontrunning censorship, if Tx_2 is intentionally added to Mem_P^h later than Tx_1, such that $Ord(Tx_2) < Ord(Tx_1)$, and Tx_2 is eventually included into a block.

$$\begin{aligned} &Tx_1 \in Mem_P^{h,t} \wedge Tx_2 \notin Mem_P^{h,t} \wedge \\ &\quad \exists t' > t, h' > h : Tx_1, Tx_2 \in Mem_P^{h,t'} \wedge \\ &\quad Ord(Tx_2) < Ord(Tx_1) \wedge Tx_2 \in Inject(M) \wedge \\ &\quad Tx_2 \in B^{h'} \end{aligned}$$

This nuances the previous censorship case, by injecting a new transaction in the mempool in *front* of others with the purpose of delaying, instead of modifying the existing transactions order by deviating from Alg_{ord}. It is enough that the injected transaction be included in a block to consider the action as censoring, the ordering pushing the censored transaction further than its normal position. We note that the insertion of Tx_2 in this action can be generalised to a set of transactions, resulting in added delay.

Transaction Censorship Observability (CObs). We complement the above actions by an informal property, which we consider essential for the design of a rollup sequencer. This property enhances the integrity guarantees offered by a TEE for code execution by enabling the detection of censorship targeting specific transactions:

A sequencer design has the CObs property if, whenever (SimpleC), (OrderC), or (FrontC) occurs, the censored transaction can be identified by a trusted party.

CObs is limited in the censorship resistant designs compared to censorship resilient designs. Therefore, the CObs property is essential to refine the distinction between these. We consider CObs to be not applicable in situations where censorship actions cannot occur.

Censorship Vectors. In our defined censorship actions, we consider the main target to be an individual transaction. The attack surface that can be used to reach this target exposes two main potential censorship vectors, as follows:

(A) The transaction mempool opens a first possible censorship vector by allowing entities that can access the pending transactions to perform all the censorship actions.
(B) A second censorship vector is by tampering with, or more generally deviating from, the logic of Alg_{ord}. This permits attempting ordering censorship.

A third censorship vector is acting on the generated blocks, before posted to L1. However, if blocks are produced by the sequencer, we find this equivalent to (A), which exposes the transactions. If blocks are produced by a builder, then we consider an integrity mechanism preventing block alteration. This is similar to L1 PBS. However, for our setting, this does not require a relay and can be provided via signing the block using a TEE at the builder's side. A fourth censorship vector is simulating a rollup state result independent of the blocks, corresponding to simple or ordering censorship. However, this would fail in an optimistic fraud proof or zk-validity challenge that is considered by design in rollup operation. Given the above, we do not consider these last two vectors further in our analysis.

Generic Facts. We observe a set of generic facts relevant to our threat model. We formulate these as propositions followed by informal proofs:

▶ **Proposition 1** (SimpleC Weak Observability). *SimpleC is not observable with certainty by an entity, unless the entity can observe the mempool of the node receiving the transaction.*

Proof. An entity aware of the transaction submission can attempt observing the transaction censorship by tracking the public block ledger. If the valid submitted transaction is absent up to a height threshold set by convenience, this indicates SimpleC, with a certain probability. ◀

▶ **Proposition 2** (Private Mempool Effects). *If the mempool is private and the transaction flow is not altered, then only the mempool owner can perform SimpleC and FrontC, and OrderC and FrontC are not observable.*

Proof. We know that the mempool is private to the owner (sequencer or builder). Therefore, no other entity can perform actions that require access to the transactions reaching it, or observe specific alterations over a legitimate mempool transaction set. ◀

▶ **Proposition 3** (Mempool inside TEE). *If the mempool runs inside a TEE that guarantees the integrity of the implementation and the transaction flow is not altered before entering the mempool, then no censorship action can be performed and CObs is not applicable.*

Proof. We assume that the mempool implementation running inside the TEE is correct in the sense of appropriate mempool maintenance. We also assume that transactions reach the mempool within the TEE via a secure communication channel. Any entity with censorship intent cannot access the transactions or the mempool content at runtime. ◀

▶ **Proposition 4** (Ordering inside TEE). *If the transaction ordering algorithm is executed inside a TEE, OrderC is not possible.*

Proof. As above, we assume the ordering implementation running inside the TEE is correct. No entity can access and alter the Alg_{ord} code at runtime, which is necessary for OrderC. ◀

▶ **Proposition 5** (OrderC Requirements). *OrderC requires a censorship perpetrator to use both censorship vectors (A) and (B).*

Proof. The entity attempting censorship requires access to the mempool to target the specific transactions for OrderC. The mempool access is needed to observe the mempool state to decide when to attempt OrderC. However, this is not sufficient to change the order in a block. The censor also needs to deviate from the ordering algorithm logic, as also stated above. ◀

General Security Assumptions. In our threat model, we consider the sequencer and the builders untrusted. We assume that they will try to exploit the vectors above in all system designs that we analyse in Section 4. We do not consider exploits by third parties over the communication channels between the various components, i.e., we consider all communication takes place through secure channels such as TLS. Also, transactions cannot be altered individually, being signed by their senders.

We consider that at height h, in all designs, a trusted party, either within the system or outside, can provide an invalidity proof for a transaction Tx, denoted $Inv_P(Tx)$, that $Tx \in Inv^h$. This is necessary to distinguish between Tx censorship and not including Tx in Mem_P due to its invalidity. We note that $Inv_P(Tx)$ can be provided by any party that can access the public state of the rollup and can simulate Tx execution at height h [10].

In particular, for sound observability of FrontC only, we assume that the sequencer or the builder cannot collude with or impersonate users submitting transactions. However, we note that this assumption can be relaxed, under most common rollup dynamics, a case that we defer for discussion in Appendix A.1.

Concerning TEEs integration, we assume that any code executed within TEEs is attested by an off-band mechanism, e.g., DCAP attestation [5]. Generically, we assume that private credentials can be generated within TEEs and bound to their identity as part of the attestation [24]. These credentials can be used to sign and verify TEE produced output.

4 Generic PBS and Non-PBS Sequencer Designs

In this section, we present several generic variations of sequencer designs: some monolithic (non-PBS) and others that obtain blocks from a separate builder (PBS). In a distributed setting with multiple builders, a variation is how blocks (and therefore, the providing builder) are selected at runtime. Typical selection algorithms simply choose the most profitable blocks. This diminishes the impact on preventing censorship provided by running a distributed builder set, irrespective of the percentages of honest and malicious builders. As long as a malicious builder censors transactions and prioritises a block to be picked by the sequencer, the difference is made by anti-censorship measures included by the builder and sequencer designs. In these designs, we find that TEEs play a fundamental role. Therefore, we focus on the sequencer-builder interaction, representative for the analysis of the censorship actions considered in Section 3. We start from the typical nonsecured monolithic sequencer, and we advance through several designs up to a novel variant that we propose in Section 5.

4.1 Non-PBS Designs

In what follows, we describe and analyse the non-PBS designs.

Non-TEE Baseline Monolithic Sequencer

This design implies that all components of a sequencer, including block building, are deployed and executed on the same node. We consider it just for reference purposes, as it is the typical design used in a sequencer implementation.

Since only the sequencer can observe the transactions in the private mempool, Prop. 2 (Private Mempool Effects) implies that only the sequencer can perform SimpleC and FrontC. As the sequencer can exploit both censorship vectors (the mempool and the transaction ordering algorithm), Prop. 5 (OrderC requirements) implies that it can also perform OrderC.

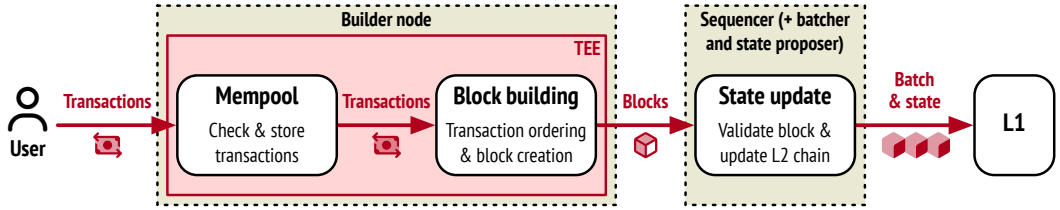

Figure 2 High level overview of PBS interaction with a TEE-enabled builder, which directly receives the transaction flow.

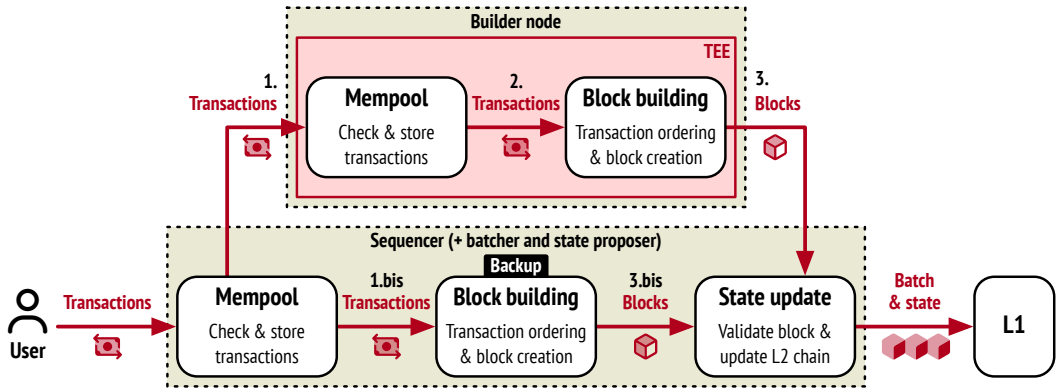

Figure 3 High level overview of PBS interaction with a TEE-enabled builder, when the transaction flow runs via the sequencer.

Concerning the observability property CObs: as stated in Prop. 1 (SimpleC Weak Observability), a user cannot observe with certainty SimpleC. Prop. 2 establishes that OrderC and FrontC cannot be observed.

TEE-Enabled Monolithic Sequencer

In this design, the sequencer is also monolithic and the potential censor, but the sequencer implementation runs inside a TEE. Since the mempool and the ordering algorithm run in a TEE, using Prop. 3 (Mempool inside TEE) we conclude that SimpleC, OrderC, and FrontC are not possible. As a result, the censorship observability property does not apply.

4.2 PBS Designs

We follow with two generic variations of PBS, where we consider running either the complete builder or both the builder and the sequencer within a TEE to protect against censorship.

Non-TEE Sequencer and TEE-enabled Builder

In this design, block construction is the attribute of the builder, which maintains the private mempool within the TEE secured space. There are two variations of this design:

(I) Transactions are sent directly to the builder, which produces the secured block for the sequencer. This design is summarised in Figure 2. Because the builder's private mempool operates inside a TEE, Prop. 3 (Mempool inside TEE) ensures that the builder and the sequencer cannot perform any censorship action and CObs is not applicable in this case.

(II) Transactions are sent to the sequencer, and then forwarded to the builder. Figure 3 depicts the transaction flow. The sequencer can build a backup block as a failover when the builder does not provide a block. This is the design implemented by Rollup Boost [18]. However, we note that Rollup Boost does not consider the sequencer to be untrusted.
If the non-TEE sequencer must propose its own backup block, the design is equivalent to the baseline, where all actions are possible. SimpleC is not observable with certainty. When the block is received from the builder, the sequencer could first alter the transaction flow, which implies altering the builder's mempool. The sequencer can then perform SimpleC and FrontC. OrderC is not possible because this also requires access to the ordering algorithm executed within the TEE space (Prop. 5 - OrderC Requirements). Regarding CObs, SimpleC is not observable with certainty (Prop. 1), FrontC is not observable (Prop. 2 - Private Mempool Effects), and OrderC observability is not applicable.

TEE-Enabled Sequencer and Builder

This architecture combines the TEE-enabled monolithic sequencer design for the case of sending transactions to the sequencer, and the previous PBS design for the case when transactions are sent directly via the TEE-enabled builder. Therefore, similar to these situations, the censorship actions are not possible and censorship observability is not applicable.

5 TEE Order PBS Designs

In this section, we introduce a novel variation for a TEE-integrating PBS design, where, at the builder side, we restrict the TEE protection to the ordering only.

Non-TEE Sequencer and TEE Ordering in Builder

We consider the builder architecture split in two modules: an untrusted module responsible for the bulk of the block building, and a trusted module that contains the transaction ordering logic. We note that this ordering logic can be custom per builder, as long as its implementation within the TEE is re-attested and a public history of the used Alg_{ord} remains available. Therefore, the trusted computing base (TCB) is reduced only to the *orderer module*, which requires execution within a TEE. Inherently, this reduces the security risks, and can also optimize resource consumption. The orderer module maintains its own mempool implementation, separate from the builder's mempool. This can have a simplified functionality, in order to keep the TCB reduced and low TEE usage overhead. For instance, in what follows we do not consider loading the rollup accounts' state within the TEE. This limits transaction validation at the orderer, i.e., the module does not verify whether account balances cover the transaction cost.[2]

Figure 4 depicts the high-level interaction of the components in this design, as described below. Users send transactions to the orderer module. When the builder must produce the block at height h, at the request of the sequencer, it will issue a request to the secure orderer module. The orderer module will respond with two sets:

[2] This is an optional consideration that addresses the practical dimension of our design. This strengthens the formal results that follow, which would otherwise be weaker due to similar validity assumptions for both the builder and ordering components.

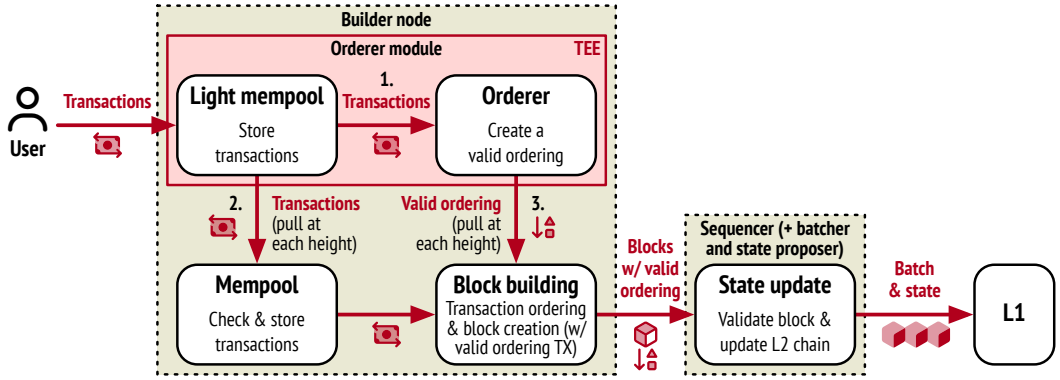

Figure 4 High level overview of PBS interactions when the builder runs TEE orderer.

- the accumulated set of transactions, which is validated and added to the builder's Mem_P^h, hence, this will be a subset of the transactions in the orderer module mempool;
- a set of ordered transaction identifiers, formed from the sender address and the transaction nonce, according to the valid ordering, signed with credentials generated within the TEE.

The builder produces the block, which includes the signed order, i.e., in an additional *valid ordering transaction* that can be verified with the TEE's public credentials, and it is submitted to the sequencer. When the block is confirmed on the L2 chain, the transactions in the block are removed from the orderer module mempool. To avoid synchronization issues, the orderer module will answer requests for a block at height h only after the block at height $h-1$ has been confirmed and after removing its respective transactions. To improve communication overhead, the transactions sent by the ordering component at each builder's request can be limited to transactions that have not been previously sent.

To the best of our knowledge, our proposed design is the first to separate exclusively the ordering component within a TEE. This allows us to extend the coverage of the CObs property compared to other designs, as we prove further below.

Censorship actions and observability analysis. Because the builder's mempool is private, Prop. 2 implies that the builder can perform SimpleC and FrontC. Because it also has access to the ordering algorithm, from Prop. 5 results that the builder can also perform OrderC. In what follows, we prove that SimpleC, OrderC and FrontC actions are observable based on the information included in blocks by the ordering component.

We denote by TXO^h the ordered set of transactions corresponding to the valid ordering transaction from a block B^h. TXO^h contains the transactions available in the mempool of the ordering component when the block B^h was finalised. We observe that $Ord(Tx)$ is consistent with the position of a transaction Tx in TXO^h. We denote by $|B^h|$ the number of transactions in block B^h, with the exception of the valid ordering transaction. We denote by $TXO^h[i..j]$ the list of transactions in TXO^h from position i to position j.

▶ **Definition 6** (Simple Censorship at Block Height h (SimpleC^h)). *A transaction Tx is subject to simple censorship at block height h, if it should have been included in the block at height h, according to Alg_{Ord} but it was not included: $Tx \notin B^h$.*

This is a particular case of SimpleC. It captures the situation when the transaction is dropped precisely from the block about to be produced. The following lemma shows how SimpleC^h can be identified using the information in B^h.

▶ **Lemma 7.** *Given a valid transaction Tx, Tx is subject to simple censorship at block height h (SimpleC^h) if and only if $Tx \notin B^h$, $Tx \in TXO^h$ and one of the two holds:*
1. *$Tx \in TXO[1..|B^h|]$ or*
2. *$|TXO^h| > |B^h|$, the block is not full and $Ord(Tx) > |B^h|$*

Proof. ($\Rightarrow$) If Tx is subject to simple censorship at block height h, according to the definition, it is not in B^h and it is in Mem_P^{h-1} (if it should have been included by the ordering algorithm). Since TXO^h includes all the transactions in Mem_P^{h-1} (the transaction in the builder's mempool during the construction of B^h), it results that $Tx \in TXO^h$. We have two possibilities: $Tx \in TXO[1..|B^h|]$ (case 1) or $Tx \notin TXO[1..|B^h|]$ (case 2), which happens if $|TXO^h| > |B^h|$ and $Ord(Tx) > |B^h|$. In this case, the block is not full. If it were, Tx would not be censored, but just absent due to reaching the block capacity.

($\Leftarrow$) We know that the set of transactions in $Mem_P^{h-1} \subseteq TXO^h$. If $Tx \in TXO[1..|B^h|]$ then Tx should precede the transaction at position $|B^h|$ in the block construction, because the transactions are sequentially included in the block according to their established order. Considering that Tx is absent from B^h, it means that in this case is subject of simple censorship at block height h. If $|TXO^h| > |B^h|$ and the block is not full, it means that there might be a set of censored transactions in $|TXO^h|$ that should have been included in the block, unless all the transactions in $|TXO^h|$ ordered after the transaction positioned at $|B^h|$ are invalid. We assumed that Tx is a valid transaction and we know that $Ord(Tx) > |B^h|$, therefore the aforementioned set includes at least Tx. ◀

▶ **Corollary 8.** *In the current design, it can be verified if a transaction that was valid at height h was subject to SimpleC if and only if the conditions in Lemma 7 hold for all $h' > h$.*

Proof. ($\Rightarrow$) We consider a transaction Tx valid at height h that is also subject to SimpleC, i.e, intentionally dropped at some height beyond h. We know that $Tx \in Mem_P^h$. From this, it means that initially $Tx \in TXO^h$. From the design, we know that Tx will be removed from $TXO^{h'}, \forall h' > h$ only when the block at height $h' - 1$ finally includes the transaction. Since Tx is subject of SimpleC, this means that it will not be present in any block $B^{h'}, h' > h$. Therefore, we know that $Tx \in TXO^{h'}, \forall h' > h$. From here, the direct proof follows similarly the two possibilities as in Lemma 7.

($\Leftarrow$) From Lemma 7 we know that when the conditions hold for a valid transaction Tx at height h, the transaction is the subject of SimpleC^h. SimpleC^h is a particular case of SimpleC. It is enough for SimpleC^h to occur at one height $h' > h$ to have the converse proof. ◀

Thus, SimpleC is observable using the information in blocks: in order to verify the conditions in Lemma 7, TXO can be extracted from the valid ordering transaction in the block and $Ord(Tx)$ is the position of Tx in TXO. Verification that the block is not full and that Tx could have been included in the block, can be done by simulating the transaction execution at the respective block height [10], which outputs its cost to be included. This is only necessary for the corner case where the censored transaction belongs to a censored group that would normally be ordered at the end of the block.

For the censorship action OrderC, we consider two separate cases:
- **(I)** ordering censorship at block level: the transactions Tx_1 and Tx_2 are in the same block
- **(II)** general ordering censorship: the transactions Tx_1 and Tx_2 are in different blocks

▶ **Lemma 9** ((I) ordering censorship in the same block)**.** *Given two valid transactions Tx_1 and Tx_2, Tx_1 is affected by OrderC in the same block iff: there exists a block B^h, $Tx_1, Tx_2 \in B^h$, $Tx_2 \prec Tx_1$, and TXO^h includes Tx_1, Tx_2, such that $Ord(Tx_1) < Ord(Tx_2)$.*

Proof. By the definition of OrderC and the fact that the transactions in $B^h \subseteq Mem_P^{h-1} \subseteq TXO^h$. ◀

▶ **Lemma 10** ((II) ordering censorship in different blocks). *Given two valid transactions Tx_1 and Tx_2, Tx_1 is affected by OrderC in different blocks iff: $\exists h$ such that $Tx_2 \in B^h$, Tx_1 is subject to simple censorship at block height h ($SimpleC^h$), $Ord(Tx_1) < Ord(Tx_2)$ and $\exists h' > h$ such that $Tx_1 \in B^{h'}$.*

Proof. We will prove only the converse implication, as the direct implication is obvious. Since Tx_1 is valid and subject to simple censorship at block height h, Lemma 7 implies $Tx_1 \in TXO^h$. Hence, Tx_1 is also in Mem_P^{h-1} because TXO^h is formed by transactions in Mem_P^{h-1} and invalid transactions, and Tx_1 is valid. Tx_1 should have been included in B^h in front of Tx_2 ($Ord(Tx_1) < Ord(Tx_2)$). But Tx_1 appears in a block after Tx_2 ($Tx_1 \in B^{h'}$), although it should have appeared before it. Hence Tx_1 is subject to ordering censorship. ◀

▶ **Lemma 11.** *Given two valid transactions Tx_1 and Tx_2, Tx_1 is subject to FrontC with high probability iff: $Tx_2 \in B^h$, $Tx_1 \in B^{h'}$, $h' \geq h$, $Tx_2 \prec Tx_1$ and $Tx_2 \notin TXO^h$, given that the builder does not collude with or impersonate the users.*

Proof. ($\Rightarrow$) From FrontC definition we can say that Tx_2 has reached Mem_P^{h-1} because it was submitted by the builder, who has exclusive access to its mempool, and the builder is interested in adding Tx_2 in front of Tx_1, therefore $Ord(Tx_2) < Ord(Tx_1)$, resulting in $Tx_2 \prec Tx_1$. Tx_2 cannot be part of TXO^h given that the builder does not collude with the users sending transactions to the ordering component. This proves the direct implication.

($\Leftarrow$) The converse implication is grounded on the fact that $Tx_2 \notin TXO^h$. This means that it was injected illegitimately by the mempool observer, i.e., by the builder to its private mempool, because all other transactions pass through the ordering component. From $Tx_2 \prec Tx_1$, it means $Ord(Tx_2) < Ord(Tx_1)$ when Alg_{Ord} decided on the order of the transactions in the block. Although we cannot determine precisely that Tx_2 was injected to the mempool *after* Tx_1, the observed ordering and not using the legal submission channel result in equivalent effects to frontrunning censorship, which can be assumed with high probability.

We note that: a) the non-collusion assumption can be relaxed in the typical dynamic of practical scenario and b) the high probability captures the effect of frontrunning, but not the intention. We further develop on these aspects in Appendix A.1. ◀

TEE-enabled Sequencer and TEE Ordering in Builder

This case combines the TEE-enabled sequencer design with the previous PBS design of the builder. In this design a replica of the transactions could also be sent to the sequencer, in order to build a block for failsafe purposes. Here, the analysis of the censorship actions and complementary properties is similar to the monolithic TEE-enabled sequencer. For the case where transactions are sent to the orderer module within the builder, the censorship action analysis is similar to the previous design.

6 Discussion

In this section, we complement our design analysis with insights into their differences. Table 1 summarizes the possible censorship actions and observability in analysed designs. For designs where both sequencer and builder may produce blocks, we only cover builder-produced blocks variation in the table. The variant of sequencer blocks in these designs serves only as a failsafe.

Table 1 Summary of non-PBS and PBS designs (Seq stands for Sequencer and Buil for Builder).

Design	SimpleC	OrderC	FrontC	CObs	Anti-Censorship
Monolithic designs (without PBS)					
No-TEE Baseline	yes	yes	yes	weak SimpleC	limited resilience
TEE Monolithic Seq	no	no	no	N/A	resistance
Designs based on PBS					
Non-TEE Seq & TEE Buil	no	no	no	N/A	resistance
TEE Seq & TEE Buil	no	no	no	N/A	resistance
Non-TEE Seq & TEE Order	yes	yes	yes	all actions	resilience
TEE Seq & TEE Order	yes	yes	yes	all actions	resilience

We know from previous work [22, 34] that PBS without using TEEs can be prone to censorship at different stages of the block building flow, as observed in L1. On the other hand, using TEEs without PBS to provide trust in the sequencer's logic can still be subject to sequencer failures and downtime, which are often documented in practice [3,19,33], leading to indistinguishable effects from transaction censorship. Therefore, we find that a mixture of TEEs with PBS is appropriate in deterring censorship in generic L2 rollup architectures.

By running the sequencer within a TEE or using PBS with TEEs, where the builder is run entirely within the secured space, the designs mainly obtain censorship-resistance. While this arguably seems the best outcome in terms of security, a malicious builder could simply shut down the TEE [35], e.g., after observing information that could lead to this decision from the prior blocks. Deciding on builders accountability for such actions and applying penalties is limited, as shutdowns are indistinguishable from censorship and other failures. This indistinguishability is strengthened by the fact that in typical L2 rollups escape hatches are designed to address both failures and censorship without distinction [2, 9, 17, 20].

Achieving censorship resilience requires precise observability of actions. Identifying censorship at transaction granularity, essentially CObs, enables resubmission of censored transactions, though without guaranteeing they will not be censored again.

Linking CObs to the particular block where transaction censorship occurs is a powerful result, since the block producer can be held accountable for the censorship action, e.g., penalised and potentially excluded temporarily from block production. In our centralised rollup setting, this is of importance in PBS designs, in the context of multiple builders.

▶ **Lemma 12.** *A design with the CObs property for OrderC and FrontC can identify the block where the censorship had effect in case of these actions.*

Proof. We observe from their definition that identifying the censored transaction in OrderC and FrontC can be linked specifically to a block at a particular height h, where the censorship action has its effect. For both actions, identifying the censored transaction Tx_1 implies identifying the censoring transactions Tx_2, for which we can determine the blocks where these are included and Tx_1 delayed. Therefore, for OrderC and FrontC, CObs can by definition pinpoint the block where the censorship occurred. ◀

▶ **Lemma 13.** *A design that can determine an uncensored view of the builder's mempool for each block height, and with the CObs property for SimpleC, can identify the block where the censorship had effect in case of this action.*

Proof. In general, we cannot pinpoint the block at which a transaction is censored in the case of SimpleC: the transaction can remain in the mempool for a span of block heights, without becoming eligible for block inclusion according to Alg_{Ord}, and be dropped at an indeterminate time during this span.

If the design can determine an uncensored view of the builder's mempool for each height from the identified transaction submission up to an eventual height h, where $Tx \notin B^h$, then the design can pinpoint the precise block height where the transaction was dropped by SimpleC. Distinguishability from dropping due to invalidity can be determined according to the general security assumptions in Section 3. Therefore, the design can link CObs for SimpleC to the block where the censorship occured. ◀

▶ **Theorem 14.** *The TEE Order designs provide linkability to the censoring block for (SimpleC), (OrderC) and (FrontC).*

Proof. By Corollary 8 and Lemmas 9, 10 and 11 the TEE Order designs provide CObs for (SimpleC), (OrderC) and (FrontC). The valid transaction ordering inside each block provides an uncensored view of the builder's mempool. The proof results from Lemmas 12 and 13. ◀

By identifying censored transactions and linking them to the censoring block, TEE Order designs provide the rollup system with the power to counter censorship attempts. Therefore, the TEE Order designs enable censorship-resilience. Defining a precise rollup protocol that leverages these designs is out of the scope of the current paper. Nevertheless, we further provide several conceptual ideas in Appendix A.2.

7 Related Work

To our knowledge, our work is the first to provide a thorough analysis of the possible censorship actions in the centralised L2 context, capturing the censoring entity's interaction with the mempool. With respect to defining censorship, Wahrstätter et al. [34] differentiate at a simplified higher level between strict and weak censorship in the context of L1, where the first represents transaction exclusion and the latter transaction delaying. In particular, [34] does not take into account the mempool view of the censor and does not consider specific ordering nuances on delaying, as we do. This latter issue is further discussed in Appendix A.3.

Chaliasos et al. [13] takes a formal approach at addressing censorship in zk-rollups, using Alloy, a declarative modelling language. This focuses on the use of escape hatch smart contract mechanisms via L1 and not on addressing censorship in L2 at the sequencer operations level. Koegl et al. [25] discuss the case when the state proposer could be independent of the sequencer and refuse to consider transactions effect in the state submitted to L1. The authors conclude as we do in §3 that this would fail due to typical rollup validations.

Rollup Boost [18] is the only sequencer solution that we are aware of, which combines the use of TEEs with PBS, deployed in practice, for the Unichain rollup [11]. It uses the non-TEE sequencer and TEE-enabled builder design described in §4 to ensure verifiable priority ordering of transactions, but assumes the sequencer to be trusted. The implementation focus is on performance, producing high frequency partial blocks for fast confirmation.

Li [28] proposes a fairness solution for L1, where each node operates a TEE secured mempool, focused on maintaining a verifiable ordering predicate. However, the solution does not use a separate lightweight TEE-secured mempool component, as in our proposed design. It requires access to the complete blockchain state and implies a preliminary trusted broadcast of the transaction to accept it into the node's mempool.

Other solutions that address the L2 sequencer censorship resort to variants of distributed protocols, assuming a trusted peers quorum. TeeRollup [35] proposes a network of sequencers, implemented using different TEEs. This represents a replicated variation of the TEE-enabled monolithic sequencer design. The motivation is to tolerate TEE corruption, including intentional crashing. This illustrates the rigidity of a design where the entire sequencer

operates within the TEE. The system does not differentiate censorship actions, using a generic slashing mechanism. To our knowledge, we are the first to propose a solution that leverages TEEs at a finer granularity, resulting in precise observability of censorship details.

Han et al. [21], propose an L1 PBS based shared sequencer for multiple rollups, which uses weighted leader election to reduce the censoring builders selection probability. Capretto et al. [12] propose decentralising the sequencer by using a protocol based on Setchain, a decentralised Byzantine fault tolerant (BFT) data structure for grow-only sets. The solution combines a sequencer node with a data availability service and prevents censorship under BFT, assuming fewer than one-third of nodes are malicious. Motepalli et al. [30] provide a SoK that explores decentralised sequencer designs, including censorship prevention as a desirable property and referring to BFT solutions for ordering guarantees and block proposer selection. While our work is set in the dominant centralised sequencer context, we consider it complementary to the decentralised approaches. These could benefit from a lightweight TEE-enabled ordering component adapted from our design to enhance censorship resilience.

8 Conclusion and Future Work

Sequencers are the primary component of rollup networks, their usual centralised implementation posing inherent censorship risks. Our work addresses this issue by leveraging TEEs. We first introduce a formal setting to capture precise censorship actions that a sequencer could exert on its private mempool. We highlight the importance of observing these actions for providing censorship resilience. We then explore TEE integration with PBS in rollups, which delegates block construction to a larger set of builder peers than the single sequencer. Using our formal framework, we analyse designs with TEEs in different spots of the PBS setting. Although integrating full node functionality with TEEs offers censorship resistance, we find this approach too rigid to observe fine-grained details of censorship attempts, like the censored transaction, the blocks where it was censored or the censoring action. In a dynamic market environment, adversarial by nature, these details can be used to apply tailored penalties and build a reputation system that promotes a robust service.

To this end, we propose a novel design which confines TEE use to protect just the ordering component in the block building process. We rigorously prove that this permits identifying a censored transaction, linking it with a specific censorship action, and to the block where the censorship occurred, directly pointing to the block creator as the censor. Although our current contribution focuses on having a solid theoretical ground, we did not ignore the practical dimension. We already took into account limiting the TEE's TCB, by restricting an initial transaction validation, and covered this in our formal proofs. although this would require a more solid probabilistic analysis. As future work, we intend to implement a prototype of the design proposed here, specifically validating it against off-the-shelf TEE technology (*i.e.*, Intel TDX or SGX) and further to perform comparisons with other implementations using similar TEEs (e.g., Rollup Boost).

References

1 AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. Accessed: September 7, 2025. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

2 Arbitrum Research: A Focus on Sequencer and Proposer Failures, 2025. Accessed: September 7, 2025. URL: https://research.arbitrum.io/t/a-focus-on-sequencer-and-proposer-failures/9533.

3 Arbitrum Sequencer Outage - Root Cause Analysis, 2025. Accessed: September 7, 2025. URL: https://dedaub.com/blog/arbitrum-sequencer-outage/.

4 Arbitrum: Timeboost, 2025. Accessed: September 7, 2025. URL: https://docs.arbitrum.io/how-arbitrum-works/timeboost/gentle-introduction.

5 Automata's DCAP Attestation v1.0.0: Building for Rollups and AI Agents, 2025. Accessed: September 7, 2025. URL: https://blog.ata.network/automatas-dcap-attestation-v1-0-0-building-for-rollups-and-ai-agents-2508221be0b8.

6 L2Beat Rollups Summary, 2025. Accessed: September 7, 2025. URL: https://l2beat.com/scaling/summary.

7 MEV-Boost in a Nutshell, 2025. Accessed: September 7, 2025. URL: https://boost.flashbots.net/.

8 OP Mainnet Chain Architecture, 2025. Accessed: September 7, 2025. URL: https://docs.optimism.io/operators/chain-operators/architecture.

9 OP Mainnet Forced Transaction Flow, 2025. Accessed: September 7, 2025. URL: https://docs.optimism.io/stack/transactions/forced-transaction.

10 Tenderly Simulator - Pending vs Historical Block, 2025. Accessed: September 7, 2025. URL: https://docs.tenderly.co/simulator-ui/pending-vs-historical-block.

11 Hayden Adams, Mark Toda, Alex Karys, Xin Wan, Daniel Gretzke, Eric Zhong, Zach Wong, Daniel Marzec, Robert Miller, Hasu, Karl Floersch, and Dan Robinson. Unichain, 2024. Accessed: September 7, 2025. URL: https://docs.unichain.org/whitepaper.pdf.

12 Margarita Capretto, Martín Ceresa, Antonio Fernández Anta, Pedro Moreno-Sánchez, and César Sánchez. Fast and Secure Decentralized Optimistic Rollups Using Setchain, 2024. Accessed: September 7, 2025. doi:10.48550/arXiv.2406.02316.

13 Stefanos Chaliasos, Denis Firsov, and Benjamin Livshits. Towards a Formal Foundation for Blockchain ZK Rollups. In *Proc. of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, CCS '25, 2025.

14 Stefanos Chaliasos, Itamar Reif, Adrià Torralba-Agell, Jens Ernstberger, Assimakis Kattis, and Benjamin Livshits. Analyzing and Benchmarking ZK-rollups. In *Proc. of the 6th Conference on Advances in Financial Technologies*, AFT '24, 2024. doi:10.4230/LIPIcs.AFT.2024.6.

15 Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX Demystified: A Top-Down Approach. *ACM Comput. Surv.*, 56(9), April 2024. doi:10.1145/3652597.

16 Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016. Accessed: September 7, 2025. URL: https://eprint.iacr.org/2016/086.

17 Francisco Gomes Figueira, Martin Derka, Ching Lun Chiu, and Jan Gorzny. A Practical Rollup Escape Hatch Design. In *Proc. of the 2025 IEEE International Conference on Blockchain and Cryptocurrency*, ICBC '25, pages 1–5, 2025. doi:10.1109/ICBC64466.2025.11114670.

18 Flashbots. Introducing Rollup-Boost - Launching on Unichain, 2024. Accessed: September 7, 2025. URL: https://writings.flashbots.net/introducing-rollup-boost.

19 Paul Ugbede Godwin. Base Outage Exposes Critical Decentralization Weaknesses, Particularly Its Reliance on a Centralized Sequencer, 2025. Accessed: September 7, 2025. URL: https://www.tekedia.com/base-outage-exposes-critical-decentralization-weaknesses-particularly-its-reliance-on-a-centralized-sequencer/.

20 Jan Gorzny, Lin Po-An, and Martin Derka. Ideal Properties of Rollup Escape Hatches. In *Proc. of the 3rd International Workshop on Distributed Infrastructure for the Common Good*, DICG '22, pages 7–12. ACM, 2022. doi:10.1145/3565383.3566107.

21 Huijian Han, Mingwei Wang, Feng Yang, Linpeng Jia, Yi Sun, and Rui Zhang. A Layer-2 Expansion Shared Sequencer Model for Blockchain Scalability. *Blockchain: Research and Applications*, page 100292, 2025. doi:10.1016/j.bcra.2025.100292.

22 Lioba Heimbach, Lucianna Kiffer, Christof Ferreira Torres, and Roger Wattenhofer. Ethereum's Proposer-Builder Separation: Promises and Realities. In *Proc. of the 2023 ACM on Internet Measurement Conference*, IMC '23, pages 406–420, 2023. doi:10.1145/3618257.3624824.

23 Maxim Jourenko, Kanta Kurazumi, Mario Larangeira, and Keisuke Tanaka. SoK: A Taxonomy for Layer-2 Scalability Related Protocols for Cryptocurrencies. Cryptology ePrint Archive, Paper 2019/352, 2019. Accessed: September 7, 2025. URL: https://eprint.iacr.org/2019/352.

24 Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating Remote Attestation with Transport Layer Security, 2019. Accessed: September 7, 2025. arXiv:1801.05863.

25 Adrian Koegl, Zeeshan Meghji, Donato Pellegrino, Jan Gorzny, and Martin Derka. Attacks on Rollups. In *Proc. of the 4th International Workshop on Distributed Infrastructure for the Common Good*, pages 25–30, 2023. doi:10.1145/3631310.3633493.

26 Maxwell Koegler. SoK: Current State of Ethereum's Enshrined Proposer Builder Separation, 2025. Accessed: September 7, 2025. doi:10.48550/arXiv.2506.18189.

27 Arad Kotzer, Daniel Gandelman, and Ori Rottenstreich. SoK: Applications of Sketches and Rollups in Blockchain Networks. *IEEE Transactions on Network and Service Management*, 21(3):3194–3208, 2024. doi:10.1109/TNSM.2024.3372604.

28 Rujia Li, Xuanwei Hu, Qin Wang, Sisi Duan, and Qi Wang. Transaction Fairness in Blockchains, Revisited. Cryptology ePrint Archive, Paper 2023/1034, 2023. Accessed: September 7, 2025. URL: https://eprint.iacr.org/2023/1034.

29 Akaki Mamageishvili, Mahimna Kelkar, Jan Christoph Schlegel, and Edward W. Felten. Buying Time: Latency Racing vs. Bidding for Transaction Ordering. In *Proc. of the 5th Conference on Advances in Financial Technologies*, AFT '23, pages 23:1–23:22, 2023. doi:10.4230/LIPIcs.AFT.2023.23.

30 Shashank Motepalli, Luciano Freitas, and Benjamin Livshits. SoK: Decentralized Sequencers for Rollups, 2023. Accessed: September 7, 2025. doi:10.48550/arXiv.2310.03616.

31 Ulysse Pavloff, Yackolley Amoussou-Guenou, and Sara Tucci-Piergiovanni. Ethereum Proof-of-Stake under Scrutiny. In *Proc. of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, pages 212–221, 2023. doi:10.1145/3555776.3577655.

32 Louis Tremblay Thibault, Tom Sarry, and Abdelhakim Senhaji Hafid. Blockchain Scaling Using Rollups: A Comprehensive Survey. *IEEE Access*, 10:93039–93054, 2022. doi:10.1109/ACCESS.2022.3200051.

33 Zoltan Vardai. Ethereum L2 Starknet Suffers 2nd Mainnet Outage in 2 Months, 2025. Accessed: September 7, 2025. URL: https://cointelegraph.com/news/starknet-outage-ethereum-l2-reliability-concerns.

34 Anton Wahrstätter et al. Blockchain Censorship. In *Proc. of the ACM Web Conference 2024*, WWW '24, pages 1632–1643, 2024. doi:10.1145/3589334.3645431.

35 Xiaoqing Wen, Quanbi Feng, Hanzheng Lyu, Jianyu Niu, Yinqian Zhang, and Chen Feng. TeeRollup: Efficient Rollup Design Using Heterogeneous TEE, 2025. Accessed: September 7, 2025. arXiv:2409.14647.

36 Fei Wu, Thomas Thiery, Stefanos Leonardos, and Carmine Ventre. Strategic Bidding Wars in On-chain Auctions. In *Proc. of the 2024 IEEE International Conference on Blockchain and Cryptocurrency*, ICBC '24, pages 503–511, 2024. doi:10.1109/ICBC59979.2024.10634354.

37 Sen Yang, Fan Zhang, Ken Huang, Xi Chen, Youwei Yang, and Feng Zhu. SoK: MEV Countermeasures. In *Proc. of the Workshop on Decentralized Finance and Security*, pages 21–30, 2024. doi:10.1145/3689931.3694911.

A Appendix

In this appendix we provide some additional insight to the following aspects: the analysis of frontrunning censorship observability, a conceptual overview of a rollup using our proposed design, the separation between re-ordering and censorship.

A.1 Frontrunning Censorship Observability

In what follows, we discuss the two aspects related to Lemma 11: a) the non-collusion assumption and b) the high probability of capturing the effect of frontrunning.

a) We note that in the proof of Lemma 11 on FrontC observability, we took into account the general security assumption that the builder does not collude with or impersonate the users, i.e., that it does not submit transactions to its TEE-secured ordering component. This limits the untrusted builder's frontrunning attempts to directly injecting transactions into its privately accessible mempool. Indeed, this limitation does not apply to designs where the builder or sequencer runs entirely within a TEE under the restriction of no possible communication channels that could leak the encrypted mempool's contents. We are aware that our assumption is strong, given the typical Ethereum context, where accounts can be created off-chain, not bound to an identity, and then used as transaction initiators. We note, however, that this assumption can hold in practice for the case of decentralised applications that require permissioned access, binding account addresses to identities.

Furthermore, we can relax this assumption if we tolerate a certain probability that our proposed designs may miss frontrunning attempts. This probability depends on the builder's pull rate of transactions from the ordering component compared to their inclusion rate within blocks. All new transactions pulled to create a new block at height h are protected from undetectable frontrunning attempts at that height h. If these transactions remain in the builder's mempool for future heights, the builder can submit frontrunning transactions to the ordering component. This situation occurs only when the transaction volume exceeds the block capacity, forcing the builder to include as many transactions as possible; otherwise, it may be detected as attempting SimpleC.

Currently, in most used rollups, the frequency of produced blocks results in underused block capacity, i.e., blocks are not full, which suggests that no transactions are kept pending over multiple heights in the mempool.[3] This results in a zero probability of not capturing frontrunning transactions in a single chosen builder's case and a low probability for the other builders in the PBS setting, considering that there is an overlap between the distinct mempool views of the builders. This could relax our assumption in a practical setting under normal transaction load. A study of the precise probability of frontrunning tolerance under high transaction load exceeds the scope of the current paper.

b) In the converse proof of Lemma 11, we cannot determine the local time when the censor injected Tx_2 in front of Tx_1. However, given the fact that the $Tx_2 \prec Tx_1$ and Tx_2 was injected via an illegal channel, i.e., not using the orderer, results in a similar effect to frontrunning censorship. There is a possible situation when the builder could have injected Tx_2 before Tx_1 was added to Mem_P^h. The proof does not capture the builder's intention to censor. However, in a practical context, there would be no reason to have Tx_2 injected in the absence of Tx_1. Therefore, we can say, with high probability, that the situation describes a censorship event.

[3] From August 7, 2025, to September 6, 2025, only 2.6% of OP Mainnet blocks, 0.06% of Base blocks, and 0% of Arbitrum blocks, were filled to more than 90% of their capacity. The precise number of full blocks requires fine grained analysis at transaction cost level between consecutive blocks.

A.2 Conceptual Description of a TEE-Order Censorship Resilient Rollup

In the centralised L2 setting, protocols may allow users to challenge public chain information, similar to optimistic rollups where proposed states and transactions can be disputed during a challenge period. In a TEE Order design, a censorship action is observable and can be tied to a specific block, enabling user challenges. Under PBS with multiple builders, such challenges identify and penalize the censoring builder, after which the user may resubmit the transaction to another builder. Since builders act individually, maintaining separate mempools, the existence of at least one honest builder ensures eventual transaction inclusion.

In order to censor a block, the centralised sequencer needs to replace the builder's signed ordering transaction included in the block. The centralised sequencer could still attempt to drop a builder's block in favour of another. This is observable by the censored builder, as in the regular case when its block is not selected according to the sequencer's agreed criterion, i.e., choosing the most profitable block. The builder can then resubmit the block or use escape hatches for its submission. Moreover, in TEE Order design, where the sequencer runs in a TEE, the integrity of the block choice should be guaranteed. Therefore, a construction as described above would provide censorship-resilience.

A.3 Re-Ordering and Censorship Distinction in L2

Related work [34] distinguishes re-ordering from censorship under the assumption that the order is established once a set of transactions is already selected for block inclusion according to the highest transaction fees, admitting, though that re-ordering may result in failing transactions. We consider this separation artificial, especially for the L2 context of a sequencer's private mempool. First, the selection based on the highest fees is nothing else than a pre-ordering of the transaction set present in the mempool, and therefore, can be considered part of the ordering. Second, in the L2 context, the sequencer typically merges the two steps, sometimes using other ordering policies like first-come first-served, optionally split on different priority lanes for transaction submission [4, 29].