Breaking the Chains of Rationality: Understanding the Limitations to and Obtaining Order Policy Enforcement

Sarisht Wadhwa
sarisht.wadhwa@duke.edu
Duke University

Luca Zanolini
luca.zanolini@ethereum.org
Ethereum Foundation

Aditya Asgaonkar
aditya.asgaonkar@ethereum.org
Ethereum Foundation

Francesco D'Amato
francesco.damato@ethereum.org
Ethereum Foundation

Fan Zhang
f.zhang@yale.edu
Yale University

Kartik Nayak
kartik@cs.duke.edu
Duke University

Abstract

Order manipulation attacks such as frontrunning and sandwiching have become an increasing concern in blockchain applications such as DeFi. To protect from such attacks, several recent works have designed order policy enforcement (OPE) protocols to order transactions fairly in a data-independent fashion. However, while the manipulation attacks are motivated by monetary profits, the defenses assume honesty among a significantly large set of participants. In existing protocols, if all participants are *rational*, they may be incentivized to collude and circumvent the order policy without incurring any penalty.

This work makes two key contributions. First, we explore whether the need for the honesty assumption is fundamental. Indeed, we show that it is *impossible* to design OPE protocols under some requirements when all parties are rational. Second, we explore the tradeoffs needed to circumvent the impossibility result. In the process, we propose a novel concept of rationally binding transactions that allows us to construct AnimaguSwap[1], the first content-oblivious Automated Market Makers (AMM) that is secure under rationality.

1 Introduction

Blockchains can provide a trustworthy platform for transacting and smart contract execution. Blockchain-powered finance applications, also known as DeFi, have grown to a market of more than $46 billion[2] in value. However, despite the strong integrity and availability properties offered by blockchains, they do not protect the *ordering* of user transactions. As a result, order manipulation attacks — e.g., frontrunning attacks, sandwich attacks — are rampant, where an attacker listens for user transactions sent in public and strategically places her exploiting transactions around the victim to gain a profit. The profits earned through inserting and reordering transactions are referred to as Maximal Extractable Values (MEV) [20]. An estimated $687m of MEV have been extracted as of the time of writing.[3]

To protect users from order manipulation attacks, an extensively explored direction [9, 11, 15, 16, 29–31, 35] is to bake in a mechanism that orders user transactions in a *data-independent* fashion, as part of the blockchain protocol or decentralized applications. Intuitively, data-independent ordering guarantees that given a set of user transactions as input, the final ordering of them on the blockchain should be independent of the transaction content. For example, some fair ordering protocols [15,29,30] order user transactions based on the *time* they are received by a *committee* of parties. Content-oblivious ordering (e.g., [9, 11, 31, 35]) guarantees that user transactions are hidden from the committee who orders them, e.g., through encryption, until after an ordering has been decided. In this case, transaction ordering may be based on any metadata, such as the ciphertext, the sender address, etc.

Both approaches can prevent an attacker from placing exploiting transactions before user transactions *after* having observed user transactions. However, all known data-independent ordering protocols share the same limitation: they only work under the strong assumption that enough parties running the protocol are honest. E.g., [29] assumes more than three-fourth of the participants are honest (for $\gamma = 1$, a parameter in their work).

Indeed, in a permissionless blockchain system where players are pseudonymous and can join and leave freely, the assumption that players are always honest is hard to justify. A much more palatable assumption is to assume rationality instead of honesty, i.e., instead of assuming parties are intrinsically honest, a rational party may take any action to maximize utility. In fact, the existence of MEV is tied to the rationality of the participants. Thus, the goal is to design a protocol so that following the protocol is incentive-compatible, which is significantly more challenging because all of the parties run-

[1] A key design in AnimaguSwap is that user orders may *transform* to a different direction—like the fictional creatures Animagi in Harry Potter—in order to achieve the desired game theoretic properties.

[2] https://defillama.com

[3] https://explore.flashbots.net/

ning the protocol may deviate from the protocols arbitrarily if doing so leads to a higher utility.

In this paper, we systematically explore the design of data-independent ordering protocols in the presence of rational parties, asking two fundamental questions:

1. All known data-independent ordering protocols require some honesty assumption. Is that a limitation of existing solutions or something fundamental? We answer this question negatively by showing an impossibility result that not only are existing protocols insecure in the presence of rational parties, but a wide range of protocols compliant to the same specification also cannot be secure.

2. Given the impossibility, what tradeoffs must one make in order to realize a data-independent ordering protocol under the rationality assumption? We explore two different tradeoffs. In the first one, we explore the use of a trusted execution environment (TEEs) to restrict the actions of the rational parties. In the second one, we propose a novel concept called *rationally binding commitments* and present the first decentralized exchange construction, called AnimaguSwap, with a built-in data-independent ordering protocol under rationality.

1.1 Overview of results

1.1.1 Existing protocols are not secure

Intuitively, it is not hard to see how rational parties might lead to an insecure execution: in existing data-independent ordering protocols [9, 11, 15, 16, 29–31, 35], there is no way to retroactively verify whether the ordering output was indeed data-independent. Thus, if violating data independence increases parties' utility, all parties running the protocol to collude is a dominant strategy.

For fair ordering protocols, if enough parties collude, they can order transactions arbitrarily by lying about when transactions are received — an action that cannot be held accountable unless assuming a global trustworthy timestamping service (which is a strong assumption for applications we care about).

For content-oblivious ordering, the situation is a bit trickier, as collusion *might be accountable*. For example, in schemes where user transactions are encrypted using threshold encryption (e.g., [16]), enough can reconstruct the decryption key if they collude. However, this way of colluding may be accountable since the decryption key itself could serve as an irrefutable proof of the fact that collusion has taken place.

This leads to a natural question for a protocol designer: can we leverage proof of collusion to design data-independent ordering protocols under rationality? Answering this question negatively and identifying the conditions under which this is true is the crux of our contribution. We observe that colluding parties do not necessarily need to decrypt the transaction or leave any proof of collusion whatsoever, by running the collusion algorithm in a way that the only outcome of collusion is a set of transactions that resemble benign user transactions while giving colluding parties a higher utility (e.g., with their frontrunning transactions inserted before the victim).

An order policy enforcement (OPE) framework and impossibilities. To prove this claim, we first present a generic framework in Section 4 that captures all known data-independent ordering protocols. Then, we show that in any concrete protocol Π following this framework, if violating the ordering policy increases parties' utility, there always exists a collusion protocol π with which parties can collude and violate the ordering policy with deniability: even after executing π, no participants of it can generate a cryptographic proof to incriminate any participants (including herself). Section 5 presents the full proof.

1.1.2 New directions informed by the impossibility

Our impossibility proof not only shows the fundamental limitations of existing approaches in terms of achieving security under rationality, but it also carves out avenues to improve. The impossibility critically relies on some assumptions about Π. First, users may go offline after sending one message (typically a transaction or a cryptographic commitment thereof). This is a desirable usability feature because users do not need to stay online. Consequently, once the user submits her transaction, the parties have the capability to retrieve it. Second, if the user sends a cryptographic commitment of her transaction, it is *binding* in that the commitment can only be opened to one transaction plaintext, which is a natural requirement so that transaction execution is unambiguous.

Designing protocols that violate these assumptions can circumvent the impossibility, but dispensing with them naïvely will lead to undesirable outcomes. For instance, if we require users to stay online, there exists a (somewhat trivial) solution where a user first sends a commitment to the parties running the ordering protocol, and then opens the commitment after the ordering is determined. This construction, while secure against collusion of parties, not only introduces a usability challenge for users but also potential problems when users refuse to open the commitment.

Using TEEs. A different approach is to curb the ability of the stakers to reconstruct the transaction before it has been committed on the chain. In this approach, we rely on a TEE such as Intel SGX to require stakers to hold (potentially shares of) transactions such that the share cannot be revealed to *anyone* before a commitment corresponding to the transaction has been committed on the chain. Thus, in effect, such a trusted hardware plays to restrict the actions available to the rational stakers.

Using rationally binding commitments. Our next result is a novel way to relax the second assumption, by introducing *rationally binding commitments*. A key observation from the impossibility proof is that if a user only sends one message and that message binds to her transaction, then if enough parties collude they can uniquely recover her transaction (and thus can frontrun it, for example), no matter what cryptographic protections are employed. (Since the user only sends one message, that message should enable recovery of some transactions; due to the binding property, the committee can recover the exact transaction the user committed to).

Can we dispense with the binding property as a way to circumvent the impossibility? This seems paradoxical. After all, a user's transaction needs to be encoded somehow in the commitment, otherwise, the commitment may be opened against her will. Our answer is to replace binding with *rationally binding*, as follows.

We first require parties running the protocol to put down collateral (i.e., to *stake*) that can be confiscated (or slashed) for detected misbehavior. We call these parties stakers hereafter. Suppose one of the stakers is designated as the "flipper" (the meaning of the name will become clear momentarily). In order to create a rationally binding commitment to a transaction tx, the user samples a random bit $\mathsf{b} \in \{0,1\}$ and depending on this bit, creates a transaction that is either the one that the user intended (tx) or a related but different transaction ($\overline{\mathsf{tx}}$), e.g., the other transaction must satisfy certain requirements that we will specify later for specific applications. The user sends b to the flipper in a deniable message [27, 36], and gets back an acknowledgment of the bit signed by the flipper. (If the flipper does not respond, the user can designate another flipper.) The user then shares the created transaction (which can be different from the one it intended) with the rest of the stakers. To open the commitment, the stakers reveal the shared transaction and the flipper reveals b, and the transaction tx will be executed. Crucially, if the flipper reveals the wrong bit $\bar{\mathsf{b}}$, the user can use the acknowledgment it received as evidence to slash the flipper.

From the user's point of view, assuming the penalty is properly setup, a rational flipper will always reveal b, so tx will always be executed, similarly to the binding property. On the other hand, from the stakers' point of view, even if all parties collude, they cannot identify which transaction will be executed, since the flipper might lie about b and there is no way for the flipper to prove the correctness of b due to the use of deniable messaging. In fact, the protocol can be made such that lying about b is a dominant strategy for the flipper by carefully crafing $\overline{\mathsf{tx}}$, which ensures that no stable collusion can be formed amongst the stakers.

In Section 6.2, we present AnimaguSwap, an Automated Market Makers (AMM) decentralized exchange that uses rationally binding commitments to defeat sandwich attacks. In our protocol, if user transaction tx sells a certain asset, then $\overline{\mathsf{tx}}$ is the reverse order, i.e., buying the same asset. If the stakers collude, they must still guess which will be executed (with no more than 1/2 probability to be right). Thus, in expectation, it is not worthwhile to attempt sandwiching.

Impossibility for OPE in the presence of binding side contracts. The intuition behind the above construction is to create distrust between the flipper and the rest of the committee because the flipper has the incentive to lie, and the committee members have no ability to verify the flipper's claims, so a stable coalition cannot be formed. This can be changed, though, if parties can set up *binding side contracts*, as follows: suppose the flipper puts down a large collateral and sets up a binding contract such that she loses the collateral if she lies about the bit. Then, although the flipper cannot prove a bit b is received from the user, it is still not rational for her to lie about it since the consequence of lying has been made dire by the flipper herself. In Section 7, we generalize this result and show that this intuition holds for a larger class of protocols when the collateral put down by parties is high enough.

Contributions

In summary, this paper makes the following contribution:

- We present a framework that captures existing protocols for data-independent order policy enforcement (OPE) such as fair ordering and content-oblivious ordering protocols.
- We present the first impossibility proof showing that a wide range of OPE protocols cannot be secure in the presence of rational parties.
- We present two ways to get around our impossibility result. First, we present an approach that uses TEEs to curb the actions available to the rational parties. Then, we propose the notion of rationally binding commitments as a practical way to circumvent the impossibility. We present the first AMM construction AnimaguSwap that can achieve data-independent ordering of user transactions in the presence of rational parties. We analyze the efficacy of AnimaguSwap using real-world blockchain data.
- Finally, we extend our first impossibility result to show a larger class of protocols cannot enforce OPE when binding side contracts are available.

2 Related Work

Data-independent ordering protocols. As reviewed in Section 1, several works purpose to order transactions independent of their content as a way to reduce MEV [20]. Below is a non-exhaustive list of protocols that are covered by the framework (Section 4) and the impossibility theorem (Theorem 1).

The first category of protocols is fair ordering. Kelkar *et al.* [30] investigate a notion of *fair transaction ordering* for (permissioned) consensus protocols, which prevents adversarial manipulation of the ordering of transactions. The authors then formulate a new class of consensus protocols, called Aequitas, that achieve fair transaction ordering while also providing the usual consistency and liveness. Their findings have been later extended in permissionless settings [28]. Subsequently, Kelkar *et al.* [29] devised Themis, a (permissioned) consensus protocol that, along the same lines as [30], achieves fair transaction ordering while preventing a liveness issue in Aequitas. Cachin *et al.* [15] introduce a *differential order fairness* property and present a quick order-fair atomic broadcast protocol that guarantees payload message delivery in a differentially fair order. The protocol of Cachin *et al.* results in a more efficient protocol than the previous solutions, but it relies on a weaker form of validity property.

The second category of solutions is content-oblivious ordering. A popular idea (used by, e.g., [9, 11, 16, 35]) is to encrypt user transactions using a threshold public key encryption scheme so that the ordering of transactions is done based on the ciphertext. Fino [31] efficiently integrates threshold encryption and secret sharing to a DAG-based BFT protocol. Shutter, Osmosis, and Sikka [9, 16, 35] are examples of operational systems in this category.

The protocols in these works make the assumption of honest majority participation, e.g., a majority (or two-thirds) of the participants do not deviate from the specified protocol, even if such deviations are undetectable. Our work investigates ways to relax such assumptions.

MEV mitigation leveraging rationality. Platforms have emerged to auction off the opportunities to extract MEV so that MEV extraction is democratized [6, 22]. MEV auctions rely on the rationality of bidders (also known as builders) to maximize MEV extraction. Our solution (in Section 6) aims to achieve a different goal of reducing MEV.

Heimbach *et al.* [25] analyzed the sandwich game between an AMM trader and predatory bots and identified the optimal slippage tolerance a trader could set to disincentive bots from attacking while limiting the probability of execution failures. Their algorithm, however, crucially relies on estimating the execution failure probability using historical data, and thus cannot guarantee accuracy. Our solution is fundamentally different and does not have this limitation.

PROF [10] is a protocol that leverages the profit-maximizing nature of proposers to promote the inclusion of fairly ordered transactions (PROF defines fairness broadly as any order that follows a given policy). PROF is agnostic to specific transaction ordering protocols and thus is complementary to our solution. Note that PROF does not address the security of ordering protocols under rationality, though it suggested a TEE-based content-oblivious ordering protocol.

Lower bounds on MEV mitigation. Ferreira *et al.* [38] presents an impossibility result showing that for a class of liquidity pool exchanges (e.g., Uniswap), for any data-dependent ordering policy (called sequencing rule in [38]), there are always valid sequences in which the miners get risk-free profits. Their result leaves it open whether data-independent ordering policies can be enforced, which is our focus. (We show it is impossible in Theorem 1.)

Data dependent ordering policies. All of the discussion in this work is only pertinent to ordering policies that are *data independent*, i.e., policies that only rely on the metadata related to the transactions and not the transaction content themselves. [38] proposes a *data-dependent* sequencing rule that alternates between BUY and SELL orders, to guarantee that user transactions are executed at a price as good as being executed at the beginning of the block (unless the miner does not gain anything from manipulating the ordering). Besides being geared towards different goals, our TEE-based protocol (Section 6.1) is application-agnostic, while [38] is AMM-specific, as is AnimaguSwap (Section 6.2); moreover, [38] relies on the assumption that each block is created by a different miner, a questionable assumption in today's Ethereum ecosystem with Proposer-Builder Separation (PBS) [14], which our solutions (Sections 6.1 and 6.2) avoid.

3 Model

We consider a set of n *users* $\mathcal{U} = \{\mathsf{u}_1, \ldots, \mathsf{u}_n\}$ that can send transactions to a ordering protocol Π. We also define some users as *eternal*, which are always online, and some as *ephemeral*, which may not stay online for long. Among the eternal users, we furthermore consider a fixed $N \leq n$ *stakers* $\mathcal{S} = \{s_1, \ldots, s_N\}$ that communicate with each other and participate in Π. We assume each staker s_i to be *rational*, i.e., s_i has a utility function to maximize. To keep things simple, we assume that this utility function is the amount of monetary profit (in the number of *tokens*) that the user can make. If a staker s_i fails to serve the role assigned to it or tries to deliberately deviate from the protocol, i.e., s_i is *Byzantine*, and a proof of this misbehavior is given, it loses a part of its stake (s_i gets *slashed*) and it might be removed from the system. A protocol specifies rules that provide rewards to stakers who complete certain tasks. We sometimes refer to users (and stakers) as *players* or *parties*.

We refer to state as a database where unconfirmed transactions (or commitments to the transactions) are buffered when received through a broadcast channel. Each user will have a different view of the state.

Notation. We denote the evaluation of a protocol using $(pub_o; (y_1, \ldots, y_k) \leftarrow \mathsf{prot}(pub_i; (x_1, \ldots, x_k))$. Here, there is a public input pub_i and private inputs $(x_1, \ldots, x_k)$, resulting in a public output pub_o and private outputs $(y_1, \ldots, y_k)$. Public inputs/outputs might be omitted if not applicable.

4 A Framework for Order Policy Enforcement

To study the common features of order policy enforcement (OPE) protocols [9, 11, 15, 16, 29–31, 35], we first present an abstract framework to capture the essence of aforementioned protocols with four subprotocols ($\mathsf{submit}, \mathsf{process}, \mathsf{order}, \mathsf{reveal}$) and two predicates

```
Framework for Order Policy Enforcement
Initialization:
1: Each staker $s_i$ runs init (possibly interactively with other stakers) to get $\mathsf{param}_i := (\mathsf{spri}_i, \mathsf{spp}_i)$
2: Each staker $s_i$ publishes $\mathsf{spp}_i$
3: Each staker $s_i$ sets $\mathsf{state}_i := \emptyset$
Transaction submission:
4: Whenever initiated by a user $u$, stakers in $\mathcal{S}$ and $u$ run (possibly interactively)
       $(\mathsf{txid}; (\perp, \mathsf{out}_1, \ldots, \mathsf{out}_N)) \leftarrow \mathsf{submit}(\mathsf{tx}, \mathsf{inp}_1, \ldots, \mathsf{inp}_N)$
   where $\mathsf{tx}$ is user's input (her transaction) and $\mathsf{inp}_i$ is staker $s_i$'s input derived from $\mathsf{param}_i$ and $\mathsf{state}_i$.
5: Each staker $s_i$ processes the metadata information and the transaction information and updates its state.
       $(\mathsf{md}_i, \mathsf{data}_i) \leftarrow \mathsf{process}(\mathsf{txid}, \mathsf{out}_i, \mathsf{state}_i)$
       $\mathsf{state}_i \leftarrow \mathsf{state}_i.\mathsf{add}((\mathsf{txid}, \mathsf{md}_i, \mathsf{data}_i))$
Transaction inclusion:
6: Whenever $\mathsf{ShouldRelease}(s_i)$ , stakers in $\mathcal{S}$ evaluate
       $(\mathsf{tSeq} = (\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell); (\mathsf{state}_1, \ldots, \mathsf{state}_N)) \leftarrow \mathsf{order}(\mathsf{state}_1, \ldots, \mathsf{state}_N)$
   where the order of $(\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell)$ is dependent only on $\mathsf{md}_1, \ldots, \mathsf{md}_\ell$.
7: Staker $s_i$ adds tSeq to the blockchain.
Transaction revealing:
8: For each $k \in [\ell]$, when $\mathsf{ShouldReveal}(\bar{\mathsf{tx}}_k)$, stakers evaluate
       $(\mathsf{tx}_k; (\mathsf{state}_1, \ldots, \mathsf{state}_N)) \leftarrow \mathsf{reveal}(\bar{\mathsf{tx}}_k; (\mathsf{state}_1, \mathsf{spri}_1), \ldots, (\mathsf{state}_N, \mathsf{spri}_N))$
```

Figure 1: A general framework that captures proposed ordering policy enforcement protocols [11,15,16,29–31] using four protocols (submit, process, order, reveal) and two predicates ShouldRelease, ShouldReveal.

ShouldRelease and ShouldReveal. To aid understanding, we show how existing schemes can be mapped to our framework in Appendix A.

Parties, transactions, and ordering policies. Our framework is run by *users*, who submit transactions, and *stakers* who execute the ordering protocol to order submitted transactions. Stakers' protocol can either be a component of a larger consensus protocol or a standalone protocol in parallel with the consensus (e.g., layer-2).

A transaction tx_i can be considered to consist of two parts – metadata md_i and data data_i. Metadata is part of a transaction not given to the application (i.e., a smart contract) for execution. Our framework captures a data-independent policy $\mathcal{P}$, i.e., a policy that takes as input a set of metadata (one for each transaction) and outputs one or more permutations of them, i.e., $\mathcal{P}(\mathsf{md}_1, \ldots, \mathsf{md}_\ell) \subseteq \sigma(\ell)$, where $\sigma(\ell)$ is the set of all permutations of $(\bar{\mathsf{tx}}_1, \ldots, \bar{tx}_\ell)$. Metadata $\mathsf{md}_i = (\mathsf{md}_i^1, \ldots, \mathsf{md}_i^N)$ represents the metadata for transaction tx_i across all N stakers.

Subprotocols. Sub-protocols are reactive, in that they are activated when specific conditions are met, and may execute in parallel to each other. We now describe the four subprotocols following the life cycle of a given transaction, although note that these subprotocols are reactive and may execute in parallel for different transactions.

- Stakers engage in an initialization protocol to generate a parameter $\mathsf{param} = (\mathsf{spri}, \mathsf{spp})$ that consists of secret parameters spri and public ones spp. Initialization will also set a local variable, state_i — the set of pending transactions with metadata, to $\emptyset$.

- First, to send a transaction tx to a blockchain, the user runs the submit protocol with stakers. Specifically,

  $$(\mathsf{txid}; (\perp, \mathsf{out}_1, \ldots, \mathsf{out}_N)) \leftarrow \mathsf{submit}(\mathsf{tx}, (\mathsf{inp}_i, \ldots, \mathsf{inp}_N))$$

  where inp_i and out_i are the input (output) from (to) staker s_i, and txid is an id identifying the transaction. We do not restrict how submit may be realized, e.g., it can be realized as a non-interactive protocol where the user simply encrypts the transaction under stakers' public keys (in which case $\mathsf{inp}_i = \mathsf{pk}_i$); submit may also be implemented with an interactive Multi-Party Computation (MPC) protocol where the user engages in MPC protocol with stakers (in this case inp_i might be secret). At the end of submit, each staker s_i receives some information about tx in out_i, which will be used in later protocols. Note that not all stakers may be required to participate in submit, however, a minimum of t_s is required ($1 \leq t_s \leq N$). For the stakers that do not participate, the input and output is $\perp$.

  Users are ephemeral, i.e., they may go offline after running submit, a usability feature enjoyed by most real-world systems [11, 15, 16, 29–31]. Consequently, $(\mathsf{txid}; (\perp, \mathsf{out}_1, \ldots, \mathsf{out}_N))$ together must contain enough information to recover tx, an observation that will play a critical role in our subsequent analysis. We discuss alternative protocols if this assumption does not hold in Section 6.

  We also assume w.l.o.g. that non-staker users submit their transactions before a staker adds its own, considering all the information revealed to it by the non-staking users.

- Having finished the submit protocol for a given tx, a staker runs a local process function to capture any local state to be used in later sub-protocols, e.g., the time at which tx was received. Specifically, $(\mathsf{md}_i, \mathsf{data}_i) \leftarrow \mathsf{process}(\mathsf{txid}, \mathsf{out}_i, \mathsf{state}_i)$.

- The goal of an OPE protocol is to produce blocks with transactions ordered in a desirable way. In our framework, whenever predicate $\mathsf{ShouldRelease}(s_i)$ is true, stakers will run the order protocol, with s_i being the leader if applicable, to order transactions and to output a sequence of transactions. Specifically, let $\mathsf{tSeq} = (\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell)$

$$(\mathsf{tSeq}; (\mathsf{state}_1, \ldots, \mathsf{state}_N)) \leftarrow \mathsf{order}(\mathsf{state}_1, \ldots, \mathsf{state}_N)$$

where each staker inputs its local set of pending transactions (with any metadata captured in process). The output is a sequence of transactions to be added to the blockchain and an updated local variable (e.g., with transactions added to the block removed).

Note that like in submit, not all stakers may be required to participate in order, however, a minimum of t_o is required ($1 \leq t_o \leq N$). For the stakers that do not participate, the input and output is $\perp$. These stakers would appropriately need to change state according to the on-chain published ordering of transactions.

This sub-protocol captures any multiparty computation mandated by an ordering protocol, e.g., fair ordering schemes generate the contents of the next block based on timestamps (or relative receiving orders) across all stakers.

- In some protocols, order only includes some cryptographic representation of transactions in the blockchain, and another step reveal is required to reveal the transaction plaintext so it can be executed. Whenever $\mathsf{ShouldReveal}(B)$ is true, stakers will run reveal to reveal transactions in B.

  Again, not all stakers may be required to participate in reveal, however, a minimum of t_r is required ($1 \leq t_r \leq N$).

  We use $\mathsf{tSeq} \models (\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell)$ to represent that if tSeq is posted on-chain after order then the reveal execution would correspond to $(\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell)$.

Requirements. To rule out trivial or impractical constructions, our framework makes the following assumptions.

First, we require $\mathsf{submit}(\mathsf{tx}, \cdot)$ to be binding to the given transaction tx in that if $(\_; (\_, \mathsf{out}_1, \ldots, \mathsf{out}_N)) = \mathsf{submit}(\mathsf{tx}, \cdot)$, then $(\mathsf{tx}; .) \leftarrow \mathsf{reveal}(\bar{\mathsf{tx}}; .)$. All practical blockchain systems do achieve this.

Second, we require that a submitted transaction is eventually included in the blockchain, and revealed, if applicable. This is the standard liveness property.

Third, we note that as expressed in the framework, the function $\mathsf{reveal}()$ takes as input the output of the function $\mathsf{order}()$ and the static private parameter in spri. Thus, we assume the protocols and the predicates in the interim do not affect the inputs to the function reveal, and thus the function reveal can be run any time after order (even before staker s_i adds the output block to the blockchain). This implies our framework does not apply to protocols that use cryptographic primitives that changes state of a transaction between order and reveal such as by using time-locked encryption [33] or witness encryption [23]. These primitives are not widely used due to their practical limitations (e.g., it is hard to calibrate the timeout in timelock encryption, and decrypting a timelock encrypted ciphertext requires constant computation; there is yet no practical witness encryption schemes [23]).

Examples. In Appendix A, we show that our framework can capture OPE protocols based on DKG [11, 16], secret-sharing [31], as well as fair ordering protocols [15, 29, 30].

5 Delineating Impossibility Conditions for Order Policy Enforcement

Existing data independent order policy enforcement (OPE) protocols order transactions under the assumption that a fraction (less than one-third or one-half) of stakers are Byzantine and the remaining stakers are honest. However, in practice, the motivation to introduce additional transactions, delete existing transactions, or to order transactions differently is to obtain higher monetary gains for the stakers. Thus, a model where all stakers are rational and maximizing their utility (in terms of monetary gains) captures the adversarial setting better. In this section, we analyze OPE protocols under such an adversary. In particular, we show that under some circumstances, there exists an attacking strategy where we can ensure that rational stakers *do not* follow the OPE protocol. The key challenge is in identifying the conditions under which this statement holds, and showing the resulting attacking strategy. Recalling the notations defined in Section 3, our result can be stated as follows:

Theorem 1. *Let Π be a protocol that follows the ordering policy enforcement framework (Fig. 1) to enforce a data-independent policy $\mathcal{P}$, and let $\mathcal{S}$ be the set of rational stakers executing Π. Suppose there exists a sequence of transactions $tSeq = \{t\bar{x}_1, \ldots, t\bar{x}_\ell\} \in \mathcal{P}(md_1, \ldots, md_\ell)$ with maximum utility for some input stream $((md_1, data_1), \ldots, (md_\ell, data_\ell))$. Moreover, let us assume that there exists a function* $\mathsf{extract}()$ *known to all stakers in $\mathcal{S}$ such that $tSeq' \models \mathsf{extract}(\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell) \in \mathcal{P}(md'_1, \ldots, md'_{\ell'})$ where tx_i corresponds to the* reveal *of $\bar{\mathsf{tx}}_i$, for another set of valid $md'_1, \ldots, md'_{\ell'}$; such that the utility from publishing $tSeq'$ is more than the utility from publishing tSeq. Then, Π cannot enforce $\mathcal{P}$.*

In other words, assuming MEV extraction is possible (i.e., extract exists), data-independent ordering policies cannot be enforced by protocols following the ordering policy enforcement framework defined in Fig. 1. The necessary extract function, in practice, can be an algorithm that uses a combination of techniques publicly known to stakers today and outputs the sequence that produces the highest utility.

To prove the above impossibility result we present an attacking protocol (Algorithm 1), and show that the stakers can present a different reality tSeq' where no proof of malice can be obtained.

Suppose an adversarial set of stakers $\mathcal{A}$ ($|\mathcal{A}| \geq \max(t_s, t_o, t_r)$, such that $\mathcal{A}$ is able to run submit, order, reveal) want to

Algorithm 1 Protocol for a set $\mathcal{A}$ of stakers extracting an ordering with a higher utility (protocol for $s_i \in \mathcal{A}$)

```
1: $\mathsf{state}^a_j \leftarrow$ if $s_j \in \mathcal{A}$ then $\mathsf{state}_j$ else $\perp$          ▷ $\mathsf{state}^a$ is a list of states $\mathsf{state}_j$ for every $\mathsf{state}_j \in \mathcal{A}$
2: $\mathsf{inp}^a_j \leftarrow$ if $s_j \in \mathcal{A}$ then $\mathsf{inp}_j$ else $\perp$          ▷ $\mathsf{inp}^a$ is a list of inputs $\mathsf{inp}_j$ for every $\mathsf{state}_j \in \mathcal{A}$
3: $\mathsf{spri}^a_j \leftarrow$ if $s_j \in \mathcal{A}$ then $\mathsf{spri}_j$ else $\perp$          ▷ $\mathsf{spri}^a$ is a list of private inputs $\mathsf{spri}_j$ for every $s_j$ in $\mathcal{A}$
4: procedure ATTACK$^{\mathsf{K}}(\mathsf{state}^a, \mathsf{spri}^a, \mathsf{inp}^a)$          ▷ Executed when $\mathsf{ShouldRelease}(s_i)$ is true
5:     $(\mathsf{tSeq} = (\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell), \mathsf{state}^a) \leftarrow \mathsf{order}(\mathsf{state}^a)$          ▷ Validators in $\mathcal{A}$ order $\ell$ transactions
6:     for $j \in \{1, \ldots, \ell\}$ do          ▷ Reveal the block earlier than protocol intended
7:         $(\mathsf{tx}_j; \mathsf{state}^a) \leftarrow \mathsf{reveal}(\bar{\mathsf{tx}}_j, \mathsf{state}^a, \mathsf{spri}^a)$
8:     $\mathsf{B} = (\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell)$
9:     $\mathsf{VerifySigs}(\mathsf{B})$
10:    $\mathsf{att\_B} \leftarrow \mathsf{extract}(\mathsf{B})$          ▷ Get MEV-extracting transactions
11:    $\mathsf{state}' \leftarrow \perp$
12:    for $\mathsf{att\_txn} \in \mathsf{att\_B}$ do
13:        $(\mathsf{txid}; (\perp, \mathsf{out}_1, \ldots, \mathsf{out}_N)) \leftarrow \mathsf{submit}(\mathsf{att\_txn}, \mathsf{inp}^a)$          ▷ Replay extracted in the desired order
14:        $\mathsf{md}_i, \mathsf{data}_i \leftarrow \mathsf{process}(\mathsf{txid}, \mathsf{out}_i, \mathsf{state}'_i)$          ▷ Add to state the MEV-extracting transactions
15:        $\mathsf{state}'_i \leftarrow \mathsf{state}'_i.\mathsf{add}((\mathsf{txid}, \mathsf{md}_i, \mathsf{data}_i))$
16:    $(\mathsf{tSeq}' = (\bar{\mathsf{tx}}'_1, \ldots, \bar{\mathsf{tx}}'_{\ell'}); \mathsf{state}') \leftarrow \mathsf{order}(\mathsf{state}')$
17:    return $\mathsf{tSeq}', \mathsf{state}'_i$          ▷ Publish the block containing the MEV-extracting transaction
```

attack, they will run Algorithm 1 using a protocol in a Trusted Execution Environment (such as Intel SGX) when $\mathsf{ShouldRelease}(s_i)$ is true (and skip the honest protocol). Such an algorithm in TEE is described in Appendix B. Note that we use an algorithm that provides deniability to the stakers. Stakers in $\mathcal{A}$ ($s_i \in \mathcal{A}$) will provide inputs to the TEE running Algorithm 1, which will release any output bit-by-bit to ensure all parties receive the output [12, Sec 5.4].

Note that all computations except the final outputs are hidden during the execution and not available to any party in the clear. Given ℓ received outputs (each one submitted by an user u_i for a transaction tx_i), and given a list spri^a of inputs spri_j of stakers $s_i \in \mathcal{A}$, an orderered list of transactions $\mathsf{tSeq} = (\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell)$ is generated (Line 5). Then, the reveal function is computed by the stakers in $\mathcal{A}$ by passing as inputs the previously generated list of ordered transactions, the list state^a of states state_i of stakers $s_i \in \mathcal{A}$, and spri^a. Once the transactions tx_i are available, transaction signatures are verified in order to confirm that each member provided the correct input to the protocol. Next, the extract function is run (Line 10) in order to introduce new transactions att_txn (Line 12), which are then submitted (Line 13) and added in the local state (Line 15). The resulting block containing MEV-extracting transactions is then published (Line 17).

At a high level, the above construction of an attacking protocol works because i) tSeq' is more profitable for the stakers than tSeq, and thus they are incentivized to join the coalition and ii) no party can prove that the coalition of stakers was formed to violate the ordering policy, and thus cannot be penalized. The following lemmas state this formally.

Lemma 1. *Let $\mathcal{A}$ denote the set of stakers participating in Algorithm 1. There exists no Π with $|\mathcal{S}| \geq 2$, if any of the following events leads to penalizing a staker:*

(i) Any user can claim, without proof, that $\mathcal{A}$ deviated from an honest execution of the protocol.

(ii) Each member $a \in \mathcal{A}'' \subseteq \mathcal{A}$ is incentivized to self-incriminate with proof, implicating themselves as part of the attack set, and thus it self-incriminates.

Proof. For i) we can see that any user can grief the set of attackers by reporting attacks without any proof. A staker will not be incentivized to participate in such a scheme.

For ii) For this, we consider the following two scenarios:

World 1. In World 1, a sequence of user identities $\{\mathsf{u}_1, \ldots, \mathsf{u}_\ell\}$ that submit transactions such that $\mathsf{tSeq} = \{\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell\}$ is a valid output as per the policy $\mathcal{P}$. A subset $\mathcal{A}''$ of the stakers run Algorithm 1 which outputs $\mathsf{tSeq}' \models \mathsf{extract}(\{\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell\}) = \{\bar{\mathsf{tx}}'_1, \ldots, \bar{\mathsf{tx}}'_{\ell'}\}$. Each transaction $\mathsf{tx}'_i \in \mathsf{tSeq}'$ is submitted by user u'_i where tx'_i is either a transaction from tSeq or a transaction involving new public keys belonging to a subset of the parties in $\mathcal{A}''$. At the end of the algorithm, each party $s_i \notin \mathcal{A}''$ has state state_i, each party $s_i \in \mathcal{A}''$ has state'_i as output by Algorithm 1. The protocol outputs tSeq'. In this world, the protocol Π failed to enforce the policy $\mathcal{P}$.

World 2. In World 2, a set of stakers $\mathcal{A}'' \setminus s_i$ generate transactions $\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell$. Now, the function extract is run on these transactions, and $\mathsf{tx}'_1, \ldots, \mathsf{tx}'_{\ell'}$ are generated. Now, while submitting these transactions, $\mathcal{A}'' \setminus s_i$ include only s_i, thus forming a set of $\mathcal{A}''$, which receive the transaction, order and reveal them in accordance to the protocol. In this world, Π successfully enforced the policy $\mathcal{P}$.

We see that the stateof all parties in both worlds are identical, and the outputs are identical. Thus, ignoring the communication between the adversarial parties in World 1 to run Algorithm 1, the worlds are indistinguishable. Thus, the incentive awarded in both worlds must also be the same. By the Lemma statement, in World 1, incriminating the attack set is a rational action; consequently, this holds in World 2 too. Since the self incrimination in World 1 leads to a loss of stake for some staker (in this case s_i), it would also lead to loss of stake in World 2. This is not a valid protocol design since any loss of stake (or slashing) can only occur with a proof of deviation from Π, whereas World 2 represents a successful enforcement of the policy $\mathcal{P}$. $\square$

With this lemma, the attacking stakers would not be incentivized to claim that they were involved in an attack and whistle-blow others involved in the process.

Lemma 2. *Assume that no user can distinguish whether any two public keys belong to the same entity except itself. Suppose there exists a sequence of transactions tSeq* $= \{\bar{tx}_1,\ldots,\bar{tx}_\ell\} \in \mathcal{P}(md_1,\ldots,md_\ell)$ *for some input stream* $((md_1,data_1),\ldots,(md_\ell,data_\ell))$*. Moreover, let us assume that there exists a function* $\mathsf{extract}()$ *known to all stakers such that* $tSeq' \models \mathsf{extract}(\mathsf{tx}_1,\ldots,\mathsf{tx}_\ell)$ *and* $tSeq' \in \mathcal{P}(md'_1,\ldots,md'_{\ell'})$ *for some input stream* $((md'_1,data'_1),\ldots,(md'_{\ell'},data'_{\ell'}))$*. Then, no user* u *can prove whether the input stream was* $((md'_1,data'_1),\ldots,(md'_{\ell'},data'_{\ell'}))$ *or some set of stakers* $\mathcal{A}'' \subseteq \mathcal{A}'$ *(with* $u \notin \mathcal{A}''$*) deviated from the protocol when the input stream was* $((md_1,data_1),\ldots,(md_\ell,data_\ell))$*.*

Proof. We cast the two scenarios in the following two worlds.

World 1. Let us consider the same World 1 as in Lemma 1.

World 2. In World 2, user identities $\{\mathsf{u}'_1,\ldots,\mathsf{u}'_{\ell'}\}$ submit transactions such that $\mathsf{tSeq}' \models \{\mathsf{tx}'_1,\ldots,\mathsf{tx}'_{\ell'}\}$ is a valid output as per $\mathcal{P}$. For each transaction $\mathsf{tx}'_i \in \mathsf{tSeq}'$ the following holds: if $\mathsf{tx}'_i \in \mathsf{tSeq}$, then the corresponding user u'_i submits it to the set of all stakers. Otherwise, some random user u'_i has a key indistinguishable from any stakers in $\mathcal{A}''$ and it submits the transactions only to $\mathcal{A}''$ (The set $\mathcal{A}''$ is enough to run submit as defined in specifications of the choice of attacker set). The protocol Π outputs tSeq' as per the policy. Moreover, each party $s_i \notin \mathcal{A}''$ has some state state''_i and $s_i \in \mathcal{A}''$ has state state'_i.

Now let us compare the two worlds:

- Each staker $s_i \in \mathcal{A}''$ has the same state state'_i in both worlds. Stakers $s_i \notin \mathcal{A}''$ may hold different states state_i and state''_i respectively. In particular, transactions that are in tSeq but not in tSeq$'$, are not a part of state''_i.
- Each transaction $\mathsf{tx}'_i \notin \mathsf{tSeq}$ are submitted to parties in $\mathcal{A}''$ in World 2 but not in World 1.
- Messages are sent to and received from a TEE as a part of executing Algorithm 1.

Other than the above differences, the two worlds are identical. To justify the first case, we observe that the same information mismatch would occur if when the user is submitting the transaction only a few of them receive the transaction (since the set of attackers $|\mathcal{A}''| \geq \max(t_s,t_o,t_r)$, and thus a transaction could be submitted to only them). Moreover, observe that the other two differences involve communication between adversarial parties which cannot be tracked by any user $u \notin \mathcal{A}''$. Thus, a user does not hold any additional information that can act as an irrefutable proof that some $\mathcal{A}'' \subseteq \mathcal{A}'$ indeed uses Algorithm 1 when the input stream is $((\mathrm{md}_1,\mathrm{data}_1),\ldots,(\mathrm{md}_\ell,\mathrm{data}_\ell))$, where $\mathcal{A}'$ were responsible for receiving, inclusion and revealing the transaction (only a subset of them may be required to attack). $\square$

With the above lemma, we know that any non-attacking staker entity does not have enough data to generate any proof of deviation from the protocol.

With both Lemmas 1 and 2, we know that in a valid protocol design, no party is incentivized to prove that a set of stakers deviated from the protocol (or does not have enough data to generate any proof).

Lemma 3. *If there exists an* $\mathsf{extract}$ *function known to stakers such that* $tSeq' \models \mathsf{extract}(\mathsf{tx}_1,\ldots\mathsf{tx}_\ell)$*, and the utility of tSeq$'$ is greater than the utility of tSeq, then publishing tSeq is strictly dominated by publishing tSeq$'$ obtained from Algorithm 1.*

Proof. A staker s_i, in order to release a transaction sequence, would choose the one that maximizes its utility. Since from lemmas 1 and 2 we have that no proof would be generated by any party, no negative incentive design can be incorporated that punishes the set of stakers $\mathcal{A}$ for following Algorithm 1. Since no staker would want to "double propose" a block, this tSeq$'$ which has a higher utility than tSeq would be published. Thus, releasing the sequence of transactions tSeq is strictly dominated by releasing tSeq$'$. $\square$

Proof of Theorem 1. From Lemma 3, we know that the staker would not publish tSeq over tSeq$'$, and thus the staker would not enforce policy $\mathcal{P}$ while following protocol Π. $\square$

6 OPE by Withholding Information

In the impossibility result in the previous section, we assumed that given a sequence of transactions tSeq, the parties have access to an $\mathsf{extract}()$ function that provides a higher utility. For existing systems such as Ethereum, such MEV extraction strategies are known for sequences of transactions such as sandwich attacks [39], frontrunning [21,32], arbitrage [21] etc. To make them work in the attack in Algorithm 1 (where tSeq is available but not in the clear), we can create an $\mathsf{extract}()$ circuit that attempts all known attack strategies and applies them to tSeq, and picks the best among them to produce a new sequence tSeq$'$.

Importantly, for such an attack to work, indeed, the $\mathsf{extract}()$ function needs to have access to *all* the information about the transaction (e.g., having access to signed transactions that cannot change). What happens if some information could be withheld from the stakers? To understand this question, let us consider the following example. An ideal strategy to sandwich an AMM transaction tx:= "*Buy y tokens of A for x tokens of B with a slippage of s*" is to produce a sequence $(\mathsf{tx}^{\text{attack}}_{BUY}, \mathsf{tx}, \mathsf{tx}^{\text{attack}}_{SELL})$ so that the first attacking transaction $\mathsf{tx}^{\text{attack}}_{BUY}$ reduces the supply of token A for tx making it pay a higher price, and $\mathsf{tx}^{\text{attack}}_{SELL}$ extracts the sandwiching profit. However, if the attackers do not know if tx was a buy or a sell transaction, and indeed if it was reversed, i.e., it was selling token A for B, then using the same attack can backfire and can result in losses for the attackers.

This idea leads to two natural questions. First, can we deviate from the framework to design a scheme that withholds some information from attacking stakers? Second, can we disincentivize attacks when the information is withheld? To answer the first question, observe that we can rely on users or TEEs held by stakers to withhold some information (e.g., whether it is a BUY or a SELL transaction); this information is only revealed during the reveal phase. We discuss how to disincentivize attacks when this is possible in Section 6.1. However, such a solution either requires the user to be online or the need for additional assumptions such as TEEs in the protocol. Instead, in Section 6.2, we devise a strategy with *rationally binding commitments* by creating an information asymmetry between a specific staker F (a flipper) and other stakers. In particular, the transaction can be modified after reveal has been invoked and F is responsible to *complete* the transaction. The asymmetry of information allows a rational F to improve its own utility at the expense of other stakers if the stakers choose to sandwich it. Consequently, this disincentivizes the other stakers to attack in the first place. We call this *rationally binding* since the correctness of the transaction relies on F being rational, which is a reasonable assumption. In this world, the client needs only to monitor the chain and hold F accountable in case it observes F does not complete the transaction correctly.

6.1 OPE when Users Withhold Information

In the scenario where users are allowed to withhold some information, the protocol design can be pretty simple. The user can simply send a commitment corresponding to the transaction, and once the commitment has been committed on the chain, the user can open the commitment. If the execution order depends on the order in which the commitments are committed, then no other party has access to the transaction content until after it is ordered. In fact, this solution is similar to existing solutions [11, 31], except that we rely on the user to reveal the content instead of some "committee of stakers". Consequently, the impossibility for OPE from the previous section does not apply.

However, this construction has a major drawback. This requires users to participate in the protocol execution all the time so that they can release information at appropriate times to ensure blockchain execution can happen at all times. The resulting system is not fault-tolerant since users may not be reliable.

Escrowing information with TEEs. To address the above drawback, we propose a simple OPE protocl where users can escrow transaction content to a trusted party realized with Trusted Execution Environments (TEEs). TEEs can protect the control flow and confidentiality of user programs with hardware mechanisms. Intel SGX [8], AMD SEV [1], and Nvidia H100 [3] are some examples of TEEs that can be used to realize our protocol.

Our protocol requires stakers to be equipped with TEEs. The TEE program guarantees that the opening is revealed only if the commitment has been included and ordered, ensuring content-oblivious ordering. Specifically, the TEE program has two functions: first, when initialized for the first time, it generates a pair of keys, and returns the public key with an attestation [8], while keeping the secret key hidden. We use $(\mathsf{sk}_i, \mathsf{pk}_i)$ to denote the keys generated by staker i's TEE; second, the TEE program decrypts secret shares of user transactions upon seeing a proof that the commitment of the transaction has been included in the blockchain.

Figure 10 specifies the protocol following the framework defined in Figure 1. Now we describe the protocol. To submit a transaction tx, a user first use computes a (t, N) secret-sharing of tx, denoted $TS = (\mathsf{txss}_1, \ldots, \mathsf{txss}_n)$, where t_r is the recover threshold and N is the number of stakers. To protect the integrity of TS, she builds a Merkle tree over TS (i.e., with elements in TS as leaves), and computes the Merkle root r_{TS}. Then, for each staker s_i, she sends the encrypted opening $OP = (r_{TS}, \pi_{\text{membership}}(r_{TS}, \mathsf{txss}_i), \mathsf{Enc}(\mathsf{pk}_i, \mathsf{txss}_i))$ to staker s_i. $\pi_{\text{membership}}(r_{TS}, \mathsf{txss}_i)$ is a standard membership proof that tx_i is i-th leaf in the Merkle tree over TS. Note that the secret share is encrypted under TEE's public key, thus kept secret from stakers. Finally, she sends r_{TS} to the stakers for inclusion and ordering. Once r_{TS} is included in the blockchain, staker s_i sends OP and proof of publication [18] of r_{TS} to her TEE — e.g., for PoS protocols, a proof of publication can be a set of signatures on a block containing r_{TS}; as described above, the TEE program verifies the proof of publication, and then decrypts OP and returns txss_i. Once at least t_r stakers get results from their TEEs, they reconstruct and reveal tx.

We omit aspects that are not different from other content-oblivious ordering protocols, such as charging transaction fees and dealing with malformed transactions. The security of the above protocol follows from the integrity and confidentiality properties of TEEs, and the binding and hiding properties of Merkle trees as a cryptographic commitment scheme.

6.2 OPE when Stakers Withhold Information

In this subsection, we describe a protocol design where some information is withheld from the attackers. However, contrary to the previous subsection where users withhold it, in this design, it is withheld by a designated rational staker called flipper F. This approach, as is, only works towards mitigating, and sometimes eliminating, sandwich attacks in constant automated market makers (AMMs) like in Uniswap V2 [7]. The key intuition is that if a set of stakers choose to sandwich a transaction, the protocol design allows the flipper to use its knowledge to gain a profit at the expense of those stakers. Thus, the binding property of the transaction relies on the flipper being rational (and not behave arbitrarily).

We first provide some background on an AMM and how sandwich attacks can be performed on transactions. Then, we present our protocol design and analyze it.

6.2.1 Background

An Automated Market Maker (AMM) such as Uniswap [7], Balancer [2], and Curve [5], uses automated algorithms to facilitate decentralized exchange of assets. AMMs set prices based on a mathematical formula based on the available liquidity of a given asset. In particular, in a Constant Product Market Maker, the product of the asset amounts in the liquidity pool is kept constant. Thus, if we have an AMM with two assets X and Y with quantities r_X and r_Y respectively, then $\mathrm{r}_X * \mathrm{r}_Y = k$ holds for some fixed value of k at all times.

When a user wants to trade one asset (X) for another (Y), they must deposit an amount of the first asset $\Delta\mathrm{r}_X$ and receive an appropriate amount of the second asset $\Delta\mathrm{r}_Y$ in return. Each transaction to the AMM is charged an additional contract fee, which we represent by f. Given the products of the liquidity stays fixed, the model ensures that the constraint $(\mathrm{r}_X + (1-\mathrm{f})\Delta\mathrm{r}_X) * (\mathrm{r}_Y - \Delta\mathrm{r}_Y) = k$ holds after the transaction is executed. There exist two common swap functions that can achieve this: SwapTokensForExactTokens and SwapExactTokensForTokens. We represent SwapTokensForExactTokens with BUY and SwapExactTokensForTokens with SELL.

In the BUY, $\Delta\mathrm{r}_Y$ is constant, and a corresponding variable $\Delta\mathrm{r}_X$ is found by holding the constraint equation as $\Delta\mathrm{r}_X = \frac{\Delta\mathrm{r}_Y \mathrm{r}_X}{(\mathrm{r}_Y - \Delta\mathrm{r}_Y)(1-\mathrm{f})}$. In the SELL, $\Delta\mathrm{r}_X$ is fixed, and a corresponding $\Delta\mathrm{r}_Y$ is the variable found as $\Delta\mathrm{r}_Y = \frac{(1-\mathrm{f})\mathrm{r}_Y \Delta\mathrm{r}_X}{\mathrm{r}_X + (1-\mathrm{f})\Delta\mathrm{r}_X}$.

Given a current state of the system containing r_X and r_Y tokens, a user can estimate $\Delta\mathrm{r}_Y$ received in exchange for depositing $\Delta\mathrm{r}_X$ or estimate $\Delta\mathrm{r}_X$ to deposit in exchange for receiving $\Delta\mathrm{r}_X$. However, if the state of the system changes due to some other transactions getting executed and affecting the liquidity pool, receiving $\Delta\mathrm{r}_Y$ for depositing $\Delta\mathrm{r}_X$ is not guaranteed. Thus, the system allows the user to specify a parameter expressed as a fraction called slippage s so that the number of tokens received by the user is not exact, e.g., $\geq (1-s)\Delta\mathrm{r}_Y$. In other words, the user's transaction is specified as "Deposit $(1-\mathrm{f})\Delta\mathrm{r}_X$ of X in exchange for $\geq (1-s)\Delta\mathrm{r}_Y$ of Y".

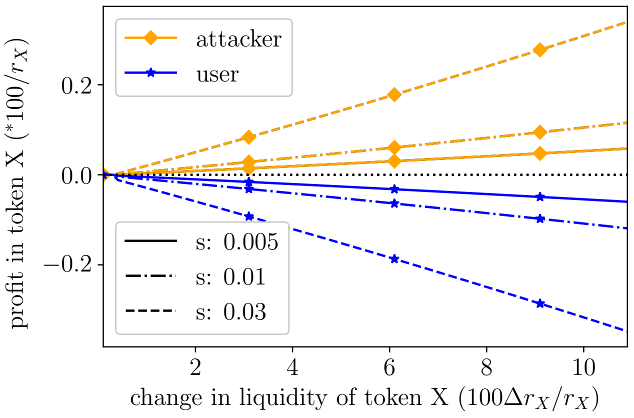

Figure 2: Gains of the attackers and users in a sandwich attack on a vanilla AMM.

6.2.2 Sandwich Attack on Constant Product AMM

The goal of a sandwich attack is to exploit the slippage constraint introduced by a user by reducing the liquidity of token Y that the user wants so that the user does not receive any more than $(1-s)\Delta\mathrm{r}_Y$ when its transaction is executed. This can be done by executing a transaction before the user's transaction is executed (frontrunning). Once the user's transaction is executed, observe that the liquidity of Y has reduced further while it is the other way around for X. Thus, the attack can then run a reversed transaction, where the attacker sells the Y earned from the frontrunning transaction, in exchange for X. Such a transaction is called backrunning, and in an AMM, the attacker obtains a higher amount of X compared to what it had deposited in the frontrunning transaction.

We refer interested readers to Appendix E for mathematical analysis of the frontrunning and backrunning transactions involved. Based on the optimal sandwich attack input from [25], we show the results graphically in Fig. 2.

6.2.3 OPE when Stakers Withhold Information

We now present a protocol that can either mitigate attacker gains, or under some parameterizations, result in attacker losses, when sandwiching is attempted. As we have seen, in the frontrunning part of a sandwich attack, the attacker reduces the liquidity of the token (token Y in our example) that the user is interested in. However, if the direction of the trade can be withheld from the attacker, then the attacker essentially has to guess one of the two directions. In situations where the attacker guesses incorrectly, it instead increases the liquidity of Y due to which the user can enjoy a much better trade and obtain $\Delta\mathrm{r}'_Y > \Delta\mathrm{r}_Y$ tokens of $\Delta\mathrm{r}_Y$.

Our protocol is shown in Figure 3. It generally follows the structure of the framework in Figure 1 except for a couple of aspects that we will describe later. Our protocol uses a designated staker F as a flipper.

```
**User intends** $\mathsf{tx}_0 = \mathbf{BUY}(Y, X, \Delta\mathbf{r}_Y, \Delta\mathbf{r}_X, \mathbf{s}, \mathbf{md})$
        $\cong \mathsf{tx}_1 = \mathbf{SELL}(X, Y, \Delta\mathbf{r}_X, \Delta\mathbf{r}_Y, \mathbf{s}, \mathbf{md})$
**On-Chain Contract** :
  var $X, Y, \Delta r_X, \Delta r_Y, s$
  payable function $\mathsf{AddMoneyT1}(\_\Delta r_X, \_s)$:
      $\mathsf{require}(\text{msg.value} == (1 + \_s)\_\Delta r_X$
      $\Delta r_X, s = \_\Delta r_X, \_s$
  payable function $\mathsf{AddMoneyT2}(\_\Delta r_Y)$:
      $\mathsf{require}(\text{msg.value} == (1 + s)\_\Delta r_Y$
      $\Delta r_Y = \_\Delta r_Y$
  function $\mathsf{AnimaguSwap}(X, Y, \text{md})$ :
      $\mathsf{tx}_{vis} = \text{BUY}(X, Y, \Delta r_X, \Delta r_Y, s, \text{md})$
      $\mathsf{tx}_\text{F} = \mathsf{NewPayment}(\text{u}, \text{F}, \text{b}(\Delta r_Y - \mathit{this}.\mathsf{amtY} + s\Delta r_Y)$
          $+(1 - \text{b})(\mathit{this}.\mathsf{amtY} - (1 + s)\Delta r_Y) - \Delta r_Y))$
      $\text{btx} = (\mathsf{tx}_{vis}, \mathsf{tx}_\text{F})$
      return btx
**Generate Transaction (All steps by User):**
        • $\text{b} = \mathsf{RandomBit}()$ // 0: Buy, 1: Sell
        • $(X, Y, \Delta r_X, \Delta r_Y, s) = \mathsf{parseParameters}(\mathsf{tx}_\text{b})$
        • $\text{md}.\mathit{hash} = \text{v}(= \mathsf{Random}())||\text{w}(= \mathsf{Random}())$
        • $\mathsf{AddMoneyT1}(\Delta r_X, s), \mathsf{AddMoneyT2}(\Delta r_Y)$
        • $\mathsf{AnimaguSwap}(X, Y, \text{md})$
**Transaction submission:**
        • User to F: $\mathsf{out}_\text{F}, \perp \leftarrow \mathsf{Enc}_{\mathsf{pk}_\text{F}}(\text{b}, \text{v}, \mathsf{txid})$
        • F to user: On receive $\mathsf{out}_\text{F}; \mathsf{out}_\text{u} \leftarrow (\text{b}, \text{v}, \mathsf{txid})_{\sigma(\mathsf{pk}_\text{F})}$
        • User: On receive $\mathsf{out}_\text{u}$; Verify $(\text{b}, \text{v}, \mathsf{txid})_{\sigma(\mathsf{pk}_\text{F})}$
        • User to $(s_1 \ldots, s_N)$ : $(\mathsf{txid}; (\perp, ss_1, \ldots, ss_N)$
          .                  $\leftarrow \mathsf{SS.share}(\text{btx}, (\mathsf{pk}_1, \ldots, \mathsf{pk}_N))$
          ▷ Where $ss_i$ represents the secret share for $s_i$
**Transaction inclusion:**
        • On receiving $ss_i$, $\mathsf{state}_i = \mathsf{append}(ss_i, \mathsf{state}_i)$
        • On $\mathsf{ShouldRelease}(\text{P} = s_i)$,
          $s_i \rightarrow \text{B} = \mathsf{ProposeBlockFrom}(\mathsf{state}_i)$
          ▷ Any block proposal mechanism
**Transaction revealing** :
        • On $\mathsf{ShouldReveal}(\text{B})$,
          $\text{B}' \leftarrow \mathsf{reveal}\left(\text{B}, \left\{\{ss_i\}_{i \in [N]}, \mathsf{out}_\text{F}\right\}\right)$
        • Add B′ to the blockchain.
```

Figure 3: AnimaguSwap specification.

Transaction generation. Let tx_{BUY} represent the transaction the user intends represented in the BUY predicate and let tx_{SELL} represent the same transaction in SELL predicate. The transactions are equivalent if no other transaction changes the state of the contract. Mathematically, $\mathsf{tx}_{SELL} = \text{SELL}(X, Y, \Delta r_X, \Delta r_Y^{MIN}, \text{md})$. We change the accepted notation slightly and represent it in terms of slippage, i.e., $\mathsf{tx}_{SELL} = \text{SELL}(X, Y, \Delta r_X, \Delta r_Y, s, \text{md})$, where Δr_Y^{MIN} can be calculated from Δr_Y and s. Similarly, $\mathsf{tx}_{BUY} = \text{BUY}(X, Y, \Delta r_X, \Delta r_Y, s, \text{md})$. Notice that the exact tokens in both cases remain Δr_X, in the former case, these are being exchanged for Y and in the latter case they are being bought in exchange for Y. We use the fact that even though the parameters to both functions are the same, they perform a very different role of buying or selling.

The user first generates a random bit b. If $\text{b} = 0$, the generated transaction $\mathsf{tx} = \mathsf{tx}_{BUY} = \text{BUY}(Y, X, \Delta r_Y, \Delta r_X, s, \text{md})$. Otherwise, $\mathsf{tx} = \mathsf{tx}_{SELL} = \text{SELL}(X, Y, \Delta r_X, \Delta r_Y, s, \text{md})$. This is the transaction that the user intends to be executed.

Now, what the user submits to a committee of stakers is not necessarily the same as what he intends to be executed. If $\text{b} = 0$, then $\mathsf{tx}_\text{b} = \mathsf{tx}$, else $\mathsf{tx}_\text{b} = \text{BUY}(Y, X, \Delta r_X, \Delta r_Y, s, \text{md})$. Since this transaction is being shared with the committee of stakers, the bit b needs to be submitted to a different staker called the flipper F.

To incentivize the flipper to not reveal the flip bit (b), the user creates another transaction tx_F which pays the flipper F some amount of tokens if the other stakers attempt to sandwich the transaction but guess the polarity incorrectly. In particular, in the example described earlier, if the user earns $\Delta r'_Y > \Delta r_Y$, then it can pay the flipper $\Delta r'_Y - \Delta r_Y$. If the example instead was a buy, and the attackers got the wrong direction of sandwich, and user paid $\Delta r'_Y < \Delta r_Y$, then it can pay the flipper additional $\Delta r_Y - \Delta r'_Y$. To represent it mathematically, the user pays the flipper $\text{b}(\Delta r'_Y - \Delta r_Y) + (1 - \text{b})(\Delta r_Y - \Delta r'_Y)$. Observe that obtaining Δr_Y is what the user was expecting in the first place; paying the remaining amount incentivizes F. In scenarios where the polarity is guessed correctly, the flipper does not gain or lose money.

Transaction submission. During the transaction submission process, the user sends the bit b to the flipper. Importantly, the user does not sign this message, ensuring that the flipper cannot prove the polarity of the transaction to the other stakers. The bit b would later be revealed by the flipper to the blockchain by sending a signed message. What if the flipper cheats and presents an incorrect value? To ensure this does not happen, the flipper sends a signed message only to the user stating that it would reveal bit b corresponding to this transaction; if the flipper does otherwise, or does not reveal any value, then it can be slashed by the user based on this message. However, one might argue that the flipper can send a similar signed message to the stakers, and the stakers can slash the flipper. In order to safeguard against that, the user sends a random value v in the unsigned message to the flipper. When returning the signed message to the user committing to b it also includes v in the commitment. In the transaction metadata for btx, a hash of $v||w$ is included. This ensures that only the user, or a party with *w* can slash the flipper, and thus the flipper is free to sign any message it wants without risk of getting slashed.

Once both these steps succeed, the user secret-shares the (potentially flipped) transaction with the remaining stakers.

Transaction inclusion and reveal. The transaction inclusion process is straightforward. An accumulator value corresponding to the transaction is added to the chain whenever ShouldRelease predicate is true. Finally, the transaction content is revealed from the secret-shares when ShouldReveal is true.[4] In this step, F reveals the bit b too so that the correct transaction is revealed.

In case the flipper reveals bit $1-\mathsf{b}$ instead of b, then the user uses the signed message $(\mathsf{b}, \mathsf{txid})_{\sigma(\mathsf{pk}_\mathsf{F})}$ to slash the flipper (not shown in the figure).

Here are a few observations related to this protocol. First, all known blockchains typically rely on accepting transactions that are signed only by the end users. This is the first protocol, to our knowledge, that includes a transaction where a portion of it (the bit b) is signed by a party (the flipper) other than the users. Second, a consequence of our approach is that, in the presence of a Byzantine flipper, the polarity of the executed transaction can be reversed. In practice, however, parties are sensitive to their utility and thus, due to the existence of the slashing mechanism, a rational flipper would always reveal the correct bit. Thus, our protocol is only *rationally binding* – this is the key aspect where we deviate from the requirement in the framework in Fig. 1. Third, since we expect the user to slash the flipper in case it deviates, the user cannot be ephemeral in the pessimistic case. However, in comparison to the protocol in the previous subsection, the blockchain execution does not need the user input for the execution to proceed. The user only needs to penalize the flipper within a reasonable timeframe (e.g., a few days). Finally, while the flipper can be any designated staker, a reasonable choice would be to have the staker that is expected to reveal the content of the transaction as the flipper. This ensures that the staker can reveal without waiting for inputs from other stakers.

6.3 Simulation Results

As an attacker trying to sandwich AnimaguSwap, there exists two non-randomized strategies, the first in which the attacker guesses $\mathsf{b}=0$, and sandwiches the transaction as it receives; or it guesses $\mathsf{b}=1$ in which case it would simulate that the flipper would release the b as 1, and flips the transaction himself to try and sandwich the resulting transaction. In order to present the results for both these pure strategies, we simulate the complete swaps on Uniswap v2 contract [7], and present the output for the expected gains for the attacker and the flipper

[4] These predicates are abstract since their choice does not affect the design. In practice, one can replace these with predicates used by Shutter DKG [16], Ferveo [11], or Fino [31].

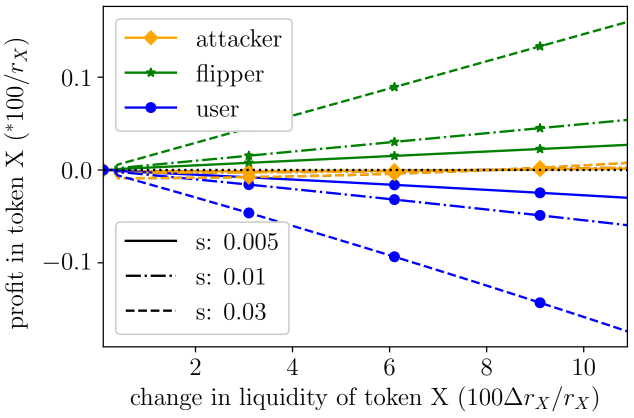

Figure 4: Gains of the attackers, flipper and the user in a sandwich attack when using AnimaguSwap (Figure 3).

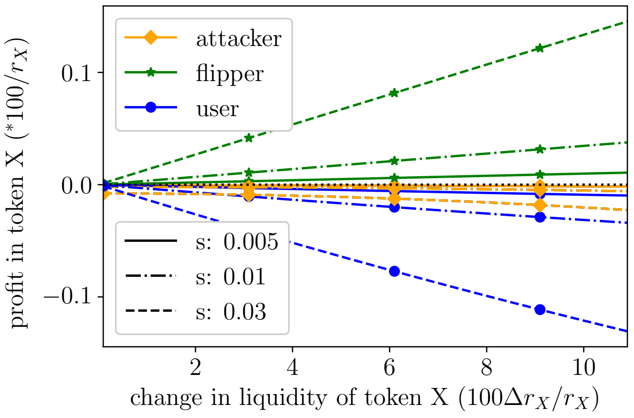

Figure 5: Gains of the attackers, flipper and the user in a sandwich attack when attacker guesses that user flipped using AnimaguSwap (Figure 3).

when the attacker decides to sandwich the transaction visible (guess $\mathsf{b}=0$) across multiple values of s set by the user in Fig. 4. Similarly, the simulation result for the case where attacker guesses $\mathsf{b}=1$ is shown in Fig. 5. For a particular randomly generated transaction between SUSHI and WETH on uniswap v2 contract (values of liquidity available pulled on June 3), the results are shown in Table 1. We also refer readers interested in the analysis for AnimaguSwap in Appendix F

7 Impossibility with Binding Side Contracts

Our protocol with rationally binding commitments works because of the distrust between different sets of stakers — in our case, the flipper and other stakers. In effect, the flipper can lie to the other stakers about the bit b and reveal a different (correct) value later. In return, the flipper would receive a utility at the expense of other stakers. What if we have a

Victim's intent	Attacker's guess	Victim's expected o/p	Victim's actual o/p	Attacker's input	Attacker's output
BUY	BUY	Get 3.65 W	Got 3.61 W	842.41 S	926.31 S
SELL	BUY	Give 4.08 W	Gave 4.03 W	842.41 S	752.68 S
SELL	SELL	Give 4.08 W	Gave 4.11 W	000.25 W	000.27 W
BUY	SELL	Get 3.65 W	Got 3.68 W	000.25 W	000.22 W

Table 1: An example swap where the Attacker sees: SELL 8592 SUSHI for WETH, when initial reserve of SUSHI is 164467.64, WETH is 73.83, with a 1% slippage. Detailed working of the example is shown in Appendix G.

mechanism to hold parties accountable for their inputs to the attacking Algorithm 1? That is, if they present different inputs to the attacking algorithm and the blockchain protocol, they could be slashed by a large amount. Indeed, in such a scenario, all parties are incentivized to present an input consistent in both the attack and the eventual blockchain protocol. In this section, we show that such an accountability mechanism can be easily implemented by creating a *binding side contract* relying on a TEE. In a nutshell, each party deposits an amount of money (the slashing amount) when submitting its transaction input in an augmented version of Algorithm 1 where a TEE containing a contract records a mapping of the party with its input. When the transaction is committed and revealed on the chain, the contract checks whether the parties' submitted input is consistent with the blockchain. If yes, the party obtains a refund; otherwise, it forfeits its deposit. Thus, the contract incentivizes the parties to attack successfully by ensuring that the amount of deposited money is larger than the gain obtained from deviating from the attacking protocol. We explain this intuition in detail, present the contract, and prove an impossibility for obtaining OPE in Appendix D.

8 Discussion and Future Work

On using primitives such as witness encryption, time lock encryption, or traceable secret sharing to circumvent Theorem 1. Our setup of Framework 1 assumes that the output of the order function is directly used as an input to the reveal function. This implies that a transaction can be revealed at any time after it is ordered so far as sufficiently many stakers participate. On the other hand, the use of cryptographic primitives such as Witness Encryption [23] and Time Lock Encryption [33] tie the reveal of transactions to satisfying some condition (e.g., the passage of time); thus, these primitives can be used to circumvent the impossibility result. The use of TEEs in Section 6.1 can be considered as an implementation of witness encryption assuming trusted hardware.

The notion of traceable secret sharing introduced by Goyal et al. [24] allows users to produce secret shares such that once the data is reconstructed, parties releasing their secret shares can be identified. However, our attack strategy in Algorithm 1 circumvents this concern by producing only the generated transactions as the output.

On sending deniable messages. In the AnimaguSwap introduced in Section 6.2, users are required to share a bit b with the flipper indicating whether the polarity of the transaction has been reversed. It is crucial to ensure the deniability of this message for users, i.e., the message sent to the flipper could also have been generated by the flipper itself. Recent studies [37] demonstrate that deniability may be compromised when Secure Guard Extensions (SGX) are present or if the flipper's keys are managed by a committee through a distributed key system. Consequently, it is necessary for users to verify that they are interacting with a single, unrestricted user as the flipper. This verification can be achieved by employing a Complete Knowledge Proof [27], which substantiates that a single user possesses unrestricted access to the information provided, thereby reinstating deniability.

On lack of knowledge of real-world entities. Our impossibility results crucially rely on the inability of the protocol participants to distinguish whether two public keys belong to the same (group of) real-world entities or not. This is reasonable, especially in a permissionless setting. However, in practice, if we are able to perform an analysis of the flow of transactions across different keys and their uses, and derive intelligence based on these transactions (e.g., [17]), we can identify the existence of such attacks with the analysis acting as a "proof".

On repeated games. In the AnimaguSwap, we explore a single-shot game where it is advantageous for one staker, the flipper, to betray other stakers. However, if the identity of the flipper is known and the game is repeated with the same flipper multiple times, it may perhaps be dominant for all parties to be colluding [26]. On the other hand, if the flipper's identity across different instances can be made unlinkable or anonymized, then betraying may again be dominant. We leave the analysis of these scenarios as future work.

Acknowledgements

This work is in part supported by an Ethereum Academic Grant.

References

[1] AMD secure encrypted virtualization (SEV). https://www.amd.com/en/developer/sev.html.

[2] Balancer. https://docs.balancer.fi/reference/math/stable-math.html.

[3] H100 tensor core GPU | NVIDIA. https://www.nvidia.com/en-us/data-center/h100/.

[4] Proof-of-stake (POS). https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[5] Understanding curve v1 curve finance. https://resources.curve.fi/base-features/understanding-curve.

[6] Mev-Boost GitHub, 2022. https://github.com/flashbots/mev-boost.

[7] Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. 2020.

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013. Issue: 7.

[9] atom_crypto. The MEV Game of the Crypto Economy: Osmosis' Threshold Encryption vs. SGX of Flashbot?, 2022. https://mirror.xyz/infinet.eth/SFjR1H1-RMnKoIoPjqkxpauVPrLYGqLHQP1dY9FHvx4.

[10] Kushal Babel, Yan Ji, Ari Juels, and Mahimna Kelkar. PROF: Fair transaction-ordering in a profit-seeking world. https://initc3org.medium.com/prof-fair-transaction-ordering-in-a-profit-seeking-world-b6dadd71f086.

[11] Joseph Bebel and Dev Ojha. Ferveo: Threshold decryption for mempool privacy in BFT networks. *Cryptology ePrint Archive*, 2022.

[12] Iddo Bentov, Yan Ji, Fan Zhang, Lorenz Breidenbach, Philip Daian, and Ari Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1521–1538, 2019.

[13] Bitcoin Wiki. Payment channels, 2021. https://en.bitcoin.it/wiki/Payment_channels.

[14] Vitalik Buterin. State of research: Increasing censorship resistance of transactions under proposer/builder separation (PBS). https://notes.ethereum.org/@vbuterin/pbs_censorship_resistance.

[15] Christian Cachin, Jovana Mićić, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *Financial Cryptography and Data Security: 26th International Conference, FC 2022, Grenada, May 2–6, 2022, Revised Selected Papers*, pages 316–333. Springer, 2022.

[16] Cducrest. Shutterized beacon chain, Mar 2022. https://ethresear.ch/t/shutterized-beacon-chain/12249.

[17] Chainalysis. Chainalysis. https://www.chainalysis.com/.

[18] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 185–200, June 2019.

[19] Joel E. Cohen. Cooperation and self-interest: Pareto-inefficiency of nash equilibria in finite random games. *Proceedings of the National Academy of Sciences*, 95(17):9724–9731, 1998. https://www.pnas.org/doi/abs/10.1073/pnas.95.17.9724.

[20] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

[21] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 910–927. IEEE, 2020.

[22] FlashBots. Flashbots resource document. https://docs.flashbots.net/, 2020.

[23] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 467–476, 2013.

[24] Vipul Goyal, Yifan Song, and Akshayaram Srinivasan. Traceable secret sharing and applications. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part III 41*, pages 718–747. Springer, 2021.

[25] Lioba Heimbach and Roger Wattenhofer. Eliminating sandwich attacks with the help of game theory. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 153–167, 2022.

[26] Lorens A Imhof, Drew Fudenberg, and Martin A Nowak. Evolutionary cycles of cooperation and defection. *Proceedings of the National Academy of Sciences*, 102(31):10797–10800, 2005.

[27] Mahimna Kelkar, Kushal Babel, Philip Daian, James Austgen, Vitalik Buterin, and Ari Juels. Complete knowledge: Preventing encumbrance of cryptographic secrets. *Cryptology ePrint Archive*, 2023.

[28] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. Order-fair consensus in the permissionless setting. *IACR Cryptol. ePrint Arch.*, 2021:139, 2021.

[29] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. *Cryptology ePrint Archive*, 2021.

[30] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.

[31] Dahlia Malkhi and Pawel Szalachowski. Maximal extractable value (mev) protection on a dag. *arXiv preprint arXiv:2208.00940*, 2022.

[32] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? *arXiv preprint arXiv:2101.05511*, 2021.

[33] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[34] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[35] Sikka inc. Sikka Projects, 2022. https://sikka.tech/projects/.

[36] Nik Unger and Ian Goldberg. Deniable key exchanges for secure messaging. In *Proceedings of the 22nd acm sigsac conference on computer and communications security*, pages 1211–1223, 2015.

[37] Ricardo Vieitez Parra et al. The impact of attestation on deniable communications. 2018.

[38] Matheus Venturyne Xavier Ferreira and David C. Parkes. Credible Decentralized Exchange Design via Verifiable Sequencing Rules. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, STOC 2023, pages 723–736, New York, NY, USA, June 2023. Association for Computing Machinery.

[39] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. High-frequency trading on decentralized on-chain exchanges. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 428–445. IEEE, 2021.

A Capturing Existing Systems in the Framework

In this subsection, we show that our framework can capture ordering policy enforcement protocols based on DKG [11,16], secret-sharing [31], as well as fair ordering protocols [15, 29, 30]. We use $(\mathsf{SS.share}, \mathsf{SS.rec})$ to denote the threshold secret sharing and reconstruction algorithms [34], and $(\mathsf{Enc}, \mathsf{Dec})$ to denote encryption and decryption.

We use $(\mathsf{SS.share}, \mathsf{SS.rec})$ to denote the threshold secret sharing and reconstruction algorithms [34], and $(\mathsf{Enc}, \mathsf{Dec})$ to denote encryption and decryption.

Protocols without an ordering policy. As a degenerate case, our framework can capture protocols that do not enforce a particular ordering policy, such as PoS Ethereum [4] (without Proposer Builder Separation [14]). Figure 6 shows the specification. submit degenerates to an identity function. process simply adds tx to state. Ethereum leaves order unspecified, as long as the output of order is a subset of state. Finally, $\mathsf{ShouldReveal}(\_)$ is always false since transactions have been revealed in the inclusion phase.

Initialization:

- Sample key pair $\mathsf{param}_i = (\mathsf{sk}_i, \mathsf{pk}_i)$ and publish pk_i.

Transaction submission:

- $\mathsf{submit}(\mathsf{tx}, \_) \to (\mathsf{H}(\mathsf{tx}); (\perp, \mathsf{tx}, \cdots, \mathsf{tx}))$ where _ denotes ignored parameters, i.e., stakers do not provide input and receive tx.
- $\mathsf{process}(\mathsf{txid}, \mathsf{tx}, \mathsf{state}_i) \to \mathsf{state}_i \cup \{(\mathsf{txid}, \mathsf{tx})\}$.

Transaction inclusion:

- Whenever $\mathsf{ShouldRelease}(s_i)$, staker s_i chooses $\mathrm{B} = (\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell)$ from state_i, and removes B from state_i. s_i adds B to the blockchain.

Transaction revealing:

- $\mathsf{ShouldReveal} \to \perp$.

Figure 6: The specification of the transaction ordering process in PoS Ethereum using our framework.

Threshold encryption based content-oblivious ordering. Content-oblivious ordering can be enforced by threshold encrypting user transactions, as in, e.g., Ferveo [11] and Shutterized Beacon Chain [16]. Figure 7 presents their specification in our framework.

At a high level, in such systems, stakers run a Distributed Key Generation (DKG) protocol to generate a key pair $(\mathsf{sk}, \mathsf{pk})$ with the secret key shared, i.e., each staker gets sk_i such that sk can be recovered from sufficiently many sk_i. After initialization, submit threshold-encrypts user transaction under pk and each staker receives the encrypted transaction. process adds the ciphertext to state. Encrypted transactions are first included in the blockchain (ordered arbitrarily in order), then the plaintext will be revealed after the block containing ciphertext is confirmed. Therefore, $\mathsf{ShouldReveal}(B)$ is true after x confirmations where x is a protocol parameter. Here we make a simplification assuming that all transactions in a block are revealed simultaneously, whereas some systems (e.g., Shutterized Beacon Chain) allow each transaction to have a different reveal time. Then, reveal is just threshold decryption by stakers.

Initialization:

- Stakers run DKG to generate $\mathsf{param}_i = (\mathsf{sk}_i, \mathsf{pk})$ where $\{\mathsf{sk}_i\}$ are secret shares of the secret key corresponding to pk. pk is published.

Transaction submission:

- $\mathsf{submit}(\mathsf{tx}, \_) \to (\mathsf{H}(\bar{\mathsf{tx}}); (\perp, \bar{\mathsf{tx}}, \cdots, \bar{\mathsf{tx}})$ where $\bar{\mathsf{tx}} = \mathsf{Enc}(\mathsf{pk}, \mathsf{tx})$, i.e., stakers do not provide input and receive encrypted tx.
- $\mathsf{process}(\mathsf{txid}, \bar{\mathsf{tx}}, \mathsf{state}_i) \to \mathsf{state}_i \cup \{(\mathsf{txid}, \bar{\mathsf{tx}})\}$.

Transaction inclusion: Whenever $\mathsf{ShouldRelease}(s_i)$ is true, order does the following

- staker s_i chooses $\mathrm{B} = \{(\mathsf{txid}_i, \bar{\mathsf{tx}}_i)\}_{i=1}^{\ell}$ from state_i,
- s_i adds $(\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell)$ to the blockchain,
- each staker s_i updates $\mathsf{state}_i = \mathsf{state}_i \setminus \mathrm{B}$.

Transaction revealing:

- $\mathsf{reveal}(\bar{\mathsf{tx}}; (\_, \mathsf{sk}_1), \cdots, (\_, \mathsf{sk}_N))$ is threshold decryption on $\bar{\mathsf{tx}}$ using $\{\mathsf{sk}_i\}_i$.

Figure 7: The specification of threshold-encryption-based content-oblivious ordering protocols using our framework.

Secret-sharing based content-oblivious ordering, e.g., Fino [31]. In these schemes, transactions are encrypted with a user-chosen key, and the key is then secret-shared with a subset of stakers (forming a so-called committee). Thus, submit secret-shares the key and sends the encrypted transaction to stakers. The rest of the steps are similar to the above threshold-encryption-based protocols. Figure 8 specifies these protocols in our framework.

Another approach is to secret-share the transaction with the stakers and obtain an accumulator value corresponding to the transaction during the submit protocol. The transaction is only revealed in the reveal phase once the accumulator value has been included on the chain and committed.

Initialization:

- Sample key pair $\mathsf{param}_i = (\mathsf{sk}_i, \mathsf{pk}_i)$ and publish pk_i.

Transaction submission:

- $\mathsf{submit}((\mathsf{tx}, \mathsf{k}), (\mathsf{pk}_1, \ldots, \mathsf{pk}_N)$ evaluates to $(\mathsf{H}(\mathsf{ct}); ((\mathsf{ct}, \mathsf{ck}_1), \ldots, (\mathsf{ct}, \mathsf{ck}_N))$ where $\mathsf{ct} = \mathsf{Enc}(\mathsf{k}, \mathsf{tx})$ and $\mathsf{ck}_i = \mathsf{Enc}(\mathsf{pk}_i, \mathsf{SS.share}(i, \mathsf{k}))$.
- For each staker s_i, $\mathsf{process}(\mathsf{txid}, (\mathsf{ct}, \mathsf{ck}_i), \mathsf{state}_i) \to \mathsf{state}_i \cup \{(\mathsf{ct}, \mathsf{ck}_i, \mathsf{false})\}$. Here false denotes ct has not been committed yet.

Transaction inclusion:

- Whenever $\mathsf{ShouldRelease}(s_i)$, staker s_i chooses a set of ℓ (a system parameter) uncommitted transactions $T = ((\mathsf{ct}_1, \_, \mathsf{false}), \ldots, (\mathsf{ct}_\ell, \_, \mathsf{false})) \subset \mathsf{state}_i$, and adds $\mathrm{B} = (\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell)$ to the blockchain.
- For each $\mathsf{ct} \in T$, replace $(\mathsf{ct}, \mathsf{ck}, \mathsf{false}) \in \mathsf{state}_i$ with $(\mathsf{ct}, \mathsf{ck}, \mathsf{true})$.

Transaction revealing: The protocol for evaluating $\mathsf{reveal}(\mathsf{ct}; (\mathsf{state}_1, \mathsf{sk}_1), \ldots, (\mathsf{state}_N, \mathsf{sk}_N))$ is:

- Each staker s_i looks up $(\mathsf{ct}, \mathsf{ck}_i, \mathsf{true})$ from state_i. s_i computes $\mathsf{k}_i = \mathsf{Dec}(\mathsf{sk}_i, \mathsf{ck}_i)$ and sends k_i to other stakers.
- All stakers compute $\mathsf{k} = \mathsf{SS.rec}(\{\mathsf{k}_i\}_i)$, remove all entries with ct from state_i, and return $\mathsf{Dec}(\mathsf{k}, \mathsf{ct})$.

Figure 8: The specification of secret-sharing-based content-oblivious ordering protocols using our framework.

Receive order fairness schemes, e.g., Themis, Aequitas, Quick Order Fairness [15, 29, 30]. Unlike the aforementioned schemes, fair ordering protocols do not attempt to hide transaction content. Instead, the guarantees are based on the time order in which a transaction is received by different stakers in the system. Thus, in our framework, during the $\mathsf{submit}()$ protocol, the user simply sends the transaction to all stakers in secure channels. The stakers then apply the $\mathsf{process}()$ function to annotate the transactions with the reception timestamp. The order protocol runs the fair ordering protocols, making use of the timestamps. (Note that some fair ordering protocols (e.g., Themis [29] and Aequitas [30]) only need relative ordering, which is simplified by timestamps.) Finally, $\mathsf{ShouldReveal}(\_)$ is always false since transactions have been revealed in the inclusion step. Figure 9 summarizes these protocols in our framework.

B Details Related to Algorithm 1

In this section, we describe a way that SGX could be used to make *Algorithm* 1 deniable. The interaction with the SGX is as follows:

Initialization:

- Sample key pair $\mathsf{param}_i = (\mathsf{sk}_i, \mathsf{pk}_i)$ and publish pk_i.

Transaction submission:

- $\mathsf{submit}(\mathsf{tx}, (\mathsf{pk}_1, \dots, \mathsf{pk}_N)) \rightarrow (\mathsf{H}(\mathsf{tx}); (\bar{\mathsf{tx}}_i, \dots, \bar{\mathsf{tx}}_N))$ where $\bar{\mathsf{tx}}_i = \mathsf{Enc}(\mathsf{pk}_i, \mathsf{tx})$.
- $\mathsf{process}(\bar{\mathsf{tx}}, \mathsf{state}, \mathsf{sk}_i) \rightarrow \mathsf{state} \cup \{(\mathsf{Dec}(\mathsf{sk}_i, \bar{\mathsf{tx}}), \tau\}$ where τ is the current time.

Transaction inclusion:

- Whenever $\mathsf{ShouldRelease}(s_i)$, staker s_i starts the order fairness protocol order with other stakers. Each staker s_i inputs state_i. Let $\mathsf{B} = (\mathsf{tx}_1, \dots, \mathsf{tx}_\ell)$.
- s_i adds B to the blockchain and sets $\mathsf{state}_i = \mathsf{state}_i \setminus \mathsf{B}$.

Transaction revealing: $\mathsf{ShouldReveal} \rightarrow \perp$.

Figure 9: The mapping of fair ordering protocols to our framework.

- Each staker initializes its SGX with the input to the MPC fed in. SGX saves this input and generates a random private key inside the SGX, and gets it remote attested, along with the code to run in *Algorithm* 1.
- All the keys are exchanged with each other, without any staker being able to link any staker key with an SGX key except its own. In addition to exchanging the key, everyone knows that each staker's respective secret shares and private inputs are committed.
- Next each SGX sends the other SGX the secret share encrypted and signed.
- Each SGX opens the received secret share from other stakers.
- Algorithm 1 is run with all inputs from each staker.
- The final return statement is revealed bit by bit, with each bit revealed is sent to other SGX, which confirm that the particular bit has been released, and the process is repeated until complete message is revealed.

The above protocol ensures deniability: We cannot link an SGX key with the staker key unless the staker itself links its own SGX key. The output from the SGX is released bit by bit, but unsigned. This ensures that no staker can claim an output was released from an SGX, thus giving no proof for the execution of the MPC Algorithm 1, since each of the SGX could be controlled by a single party simulating the MPC. The security for the MPC is ensured since no party can change inputs after the VerifySigs is executed, and can only abort at any time. At the time of aborting the process, the staker who chooses to abort has at most 1 extra bit of information than other stakers, which means the expected time to guess the value is negligibly different.

C Example Attack on DKG-based Threshold Encryption

Algorithm 2

D Impossibility in the Presence of Binding Side Contracts

The intuitive reason that a protocol similar to the one presented in Section 6.2 works is the lack of trust between different stakers. A protocol can be designed in such a way that a staker can try to cheat another staker by *lying* about the information it has. Since we remove the binding property of the transaction commit, the information for reveal is no longer cryptographically verifiable, no function in the attacker's protocol can ascertain (earlier achieved by checking the signature of transaction) whether or not the provided information is accurate. If *lying* can be made the rational action through a carefully constructed incentive design, then a protocol could be constructed where ordering is enforced rationally.

This creates a game with Pareto efficient outcomes in an inefficient Nash equilibrium [19]. However, as is well known in game theory, such games only work in absence of binding side contracts. By committing to the strategy of being *honest*, the staker claims that he would lose a huge amount in case a deviation from the strategy is observed. We show an existence of such a contract which relies on an Trusted Execution Environment (TEE). Thus, due to the rationality of the staker s_i, every other staker would get a guarantee that releasing wrong information in Algorithm 1 would not be rational for s_i. Note that each staker is confident of their own information since non-transferable proofs were provided to them by the user. Therefore, in this section we argue that even if we allow the user to create such distrust between parties by allowing multiple output blocks corresponding to the same transaction sequence output from order, it is still impossible to construct an OPE protocol in the presence of *binding side contracts*. Note that the user still only wants one of the multiple allowed output blocks and can penalize the staker that deviates from the release of the block.

To approach this impossibility, we augment the previously discussed attacker's protocol by adding rational binding property to each stakers information. In order to achieve this, we propose a binding side contract built with the help of a smart contract.

In the contract, staker s_i commits that private information $\mathsf{mpc\_inp}_i = (\mathsf{state}_i, \mathsf{spri}_i)$ (which will be used in Algorithm 1) is correct. This lets other stakers trust $\mathsf{mpc\_inp}_i$, otherwise the confiscation function could be called by some other staker and make s_i lose stake (or utility). If s_i discloses the correct information, then it receives a refund of any deposited amount.

However, if we naïvely compose the online contract described above with Algorithm 1, then Lemma 2 would no

Algorithm 2 Example attack on DKG-based threshold encryption schemes - (protocol for $s_i \in \mathcal{A}$)

```
1: $(\textsc{DKG}, \textsc{ENC}, \textsc{DEC})$ is a Distributed Key Encryption Scheme
2: $\mathsf{sk}^i, \mathsf{pk} \leftarrow \textsc{DKG}()$    ▷ $\mathsf{sk}^i = \mathsf{spri}_i$ secret share of $s_i \in \mathcal{A}$
3: $\mathsf{state}^a_j \leftarrow$ if $s_j \in \mathcal{A}$ then $\mathsf{state}_j$ else $\perp$    ▷ $\mathsf{state}^a$ is a list of states $\mathsf{state}_j$ for every $\mathsf{state}_j \in \mathcal{A}$
4: $\mathsf{inp}^a_j \leftarrow$ if $s_j \in \mathcal{A}$ then $\mathsf{inp}_j$ else $\perp$    ▷ $\mathsf{inp}^a$ is a list of inputs $\mathsf{inp}_j$ for every $\mathsf{state}_j \in \mathcal{A}$
5: $\mathsf{spri}^a_j \leftarrow$ if $s_j \in \mathcal{A}$ then $\mathsf{spri}_j$ else $\perp$    ▷ $\mathsf{spri}^a$ is a list of secret shares $\mathsf{sk}^j$ for every $s_j$ in $\mathcal{A}$
6: procedure $\textsc{Attack}^{\mathsf{K}}(\mathsf{state}^a, \mathsf{spri}^a)$    ▷ Executed when $\mathsf{ShouldRelease}(s_i)$ is true
7:     $((\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell), \mathsf{state}^a) \leftarrow \mathsf{order}(\mathsf{state}^a)$    ▷ order creates a block in descending order of fee in state
8:     for $j \in \{1, \ldots, \ell\}$ do    ▷ Reveal the block earlier than protocol intended
9:         $(\mathsf{tx}_j; \mathsf{state}^a) \leftarrow \textsc{DEC}(\bar{\mathsf{tx}}_j; \{\mathsf{state}^a, \mathsf{spri}^a\})$    ▷ reveal := DEC - Decrypt given all the secret shares $\mathsf{sk}^j$ of $s_j \in \mathcal{A}$
10:    Generate a block consisting of $\ell$ revealed transactions
11:    $\mathsf{B} = (\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell)$
12:    $\mathsf{VerifySigs}(\mathsf{B})$
13:    $\mathsf{att\_B} \leftarrow \mathsf{extract}(\mathsf{B})$    ▷ Get MEV-extracting transactions
14:    $\mathsf{state}' \leftarrow \perp$
15:    for $\mathsf{att\_txn} \in \mathsf{att\_B}$ do
16:        $(\mathsf{txid}; (\perp, \mathsf{out}_1, \ldots, \mathsf{out}_N)) \leftarrow \textsc{ENC}(\mathsf{att\_txn}, \mathsf{inp}^a)$    ▷ submit := ENC - Encrypt extracted in the desired order
17:        $\mathsf{md}_i, \mathsf{data}_i \leftarrow \mathsf{process}(\mathsf{txid}, \mathsf{out}_i, \mathsf{state}'_i)$    ▷ Add to state the MEV-extracting transactions
18:        $\mathsf{state}'_i \leftarrow \mathsf{state}'_i.\mathsf{add}((\mathsf{txid}, \mathsf{md}_i, \mathsf{data}_i))$
19:    $(\mathsf{tSeq}' = (\bar{\mathsf{tx}}'_1, \ldots, \bar{\mathsf{tx}}'_{\ell'}); \mathsf{state}') \leftarrow \mathsf{order}(\mathsf{state}')$
20:    $\mathsf{return}(\mathsf{tSeq}', \mathsf{state}')$    ▷ Publish the block containing the MEV-extracting transaction
```

Initialization: Each staker s_i load the program specified below to an TEE. Then, s_i invokes *Init*, gets pk_i and publishes pk_i along with the attestation. Users verify pk_i against the attestation before using the protocol.

Transaction submission:

$\mathsf{submit}((\mathsf{tx}, \mathsf{k}), (\mathsf{pk}_1, \ldots, \mathsf{pk}_N))$

$\forall i \in [1, N]$, compute $\mathsf{txss}_i = \mathsf{SS.share}(i, \mathsf{tx})$. Let $TS = (\mathsf{txss}_1, \ldots, \mathsf{txss}_N)$.

Build a Merkle tree over TS and denote the root as r_{TS} (which will also be the txid). Let π_i be the membership proof of txss_i.

Send $OP = (r_{TS}, \pi_i, \mathsf{Enc}(\mathsf{pk}_i, \mathsf{txss}_i))$ to each staker s_i.

$\mathsf{process}(\mathsf{txid}, OP, \mathsf{state}_i)$

Parse OP as (r, π, c). Add $(r, \pi, c, \mathsf{false})$ to state_i. Here false denotes the transaction has not been committed yet.

Transaction inclusion: Whenever $\mathsf{ShouldRelease}(s_i)$, staker s_i chooses $T = ((r_1, \_, \_, \mathsf{false}), \ldots, (r_\ell, \_, \_, \mathsf{false})) \subset \mathsf{state}_i$, and adds $\mathsf{B} = (r_1, \ldots, r_\ell)$ to the blockchain. Once B is included in the blockchain, for each $r \in B$, all stakers replace $(r, \pi, c, \mathsf{false})$ in state_i with $(r, \pi, c, \mathsf{true})$. Here true denotes the transaction with $\mathsf{txid} = r$ has been committed.

Transaction revealing: Once the Merkle root r has been committed to the blockchain, each staker s_i creates a proof of publication of r, denoted as $\pi_{\text{publication}}$. Then, s_i retrieves the corresponding membership proof $\pi_{\text{membership}}$ and the ciphertext c of the share, and invokes *Reveal*. Upon receiving txss_i from the TEE, s_i sends txss_i to other stakers. Once t_r shares are received, each staker computes $\mathsf{tx} = \mathsf{SS.rec}(\mathsf{txss}_1, \ldots, \mathsf{txss}_t)$ and outputs tx.

TEE Program run by staker i

func *Init*

$(\mathsf{sk}, \mathsf{pk}) \leftarrow\$ \mathsf{KGen}(1^\lambda)$

Seal sk to disk

return pk_i with hardware attestation.

func *Reveal*$(\pi_{\text{publication}}, r_{TS}, \pi_{\text{membership}}, c)$:

$\mathsf{txss}_i = \mathsf{Dec}(\mathsf{sk}_i, c)$

Verify $\pi_{\text{publication}}$ for r_{TS}, and $\pi_{\text{membership}}$ of tx_i w.r.t. r_{TS}

return txss_i

Figure 10: The specification of a secret-sharing-based content-oblivious ordering protocol using TEEs.

longer be true, since now the presence of $\mathsf{mpc\_inp}_i$ in a contract could in some cases be a proof that the staker released privileged information, and can be slashed. For example, in a distributed key generation based protocol, the committee member (staker) cannot reveal its share of the secret key online, or else it would be slashed. Therefore to hide the same, we make use of an oracle-based hashed time lock contract, where an TEE acts as an oracle to release a secret preimage of an on-chain hash in order to facilitate the refund or the confiscation of the amount in the contract.

Another important component to this collusion is that if the secret share is made unverifiable, i.e., it cannot be determined which among the stakers provided the incorrect input, then all involved stakers would have to lose utility. If the set of attackers is the complete set of stakers, then since everyone is losing utility, no staker would call the confiscate function for anyone. To design around this, we create a negative reward strategy in which the staker will lose staked utility unless he can prove to the TEE about the correctness of its own input, and receive a secret to publish on-chain as a proof that he was able to convince the TEE that the said input is correct.

Thus, we arrive at the contract presented in Algorithm 3. It consists of two parts, a TEE attested code and an on-chain contract. The staker creates a *remote attestation* to the code described in Algorithm 3 and that the output to the function keygen (Line 11, Algorithm 3) that generates and stores an asymmetric key inside the TEE. A secret is randomly generated inside the TEE through the function generate_hash (Line 14, Algorithm 3), which returns the hash of the secret and a signature on the hash, input and a block hash (the successor of which is being attacked) to ensure that the function was run inside the TEE. Using this signed hash value, the staker s_i now calls commit function in the contract (Line 44, Algorithm 3) and commits that she would know the value of the preimage to the hash in the future. Next, the Algorithm 1 is called, where inside the MPC, the signature of the TEE is checked (parties input previous block hash and the committed hash value on the online contract). After the MPC generates a list of transactions tSeq, s_i passes it on to the TEE by calling the update_MPC_block function (Line 19, Algorithm 3). Now any new transactions that were generated by the MPC would not have any corresponding inputs to the MPC, and thus would need to be marked as transactions that the staker did not commit information to. All the other transactions have the input committed by the staker. Whenever another transaction sequence is added to the chain, it is checked to be the successor of the current hash stored (Line 25, Algorithm 3), which is eventually used to check whether or not the MPC transaction sequence has been confirmed on-chain or not (checkConfirmed). Whenever $\mathsf{ShouldReveal}(\mathsf{tx})$ is true, the transaction would be revealed in a block B by following the procedure in the protocol. This B acts as proof that tx was released and the committed input inp has a corresponding commitment to this transaction. If the check passes then the transaction is also marked, this time because its corresponding output has been checked (Line 28, Algorithm 3). Finally, when all transactions have been marked, s_i calls get_preimage (Line 33, Algorithm 3) in which the TEE checks whether all transactions have been marked and if the check passes, the preimage to the hash is revealed. Using this secret, s_i can call the refund (Line 48, Algorithm 3) function in the contract to get back her committed amount. If on the other hand the timeout expires, then any user can call confiscate function (Line 51, Algorithm 3) to burn all the amount stored in the contract, and take a small transaction fee (ε) from the burnt amount.

Lemma 4. *Given the staker s_i provides consistent input to Augmented Algorithm 1 and at the time of* reveal*, and the same Augmented Algorithm 1 publishes a transaction sequence tSeq, s_i will receive back the amount set as collateral.*

Proof. We are given that the Algorithm 1 succeeds and publishes the transaction sequence tSeq. Any transaction in this transaction sequence could be present in state_i or not in state_i. If the transaction is not in state_i, then s_i did not commit to any information about this transaction, and is thus marked off. If the transaction (tx) was in the state_i, and it made into the transaction sequence, then this transaction would be revealed when $\mathsf{ShouldReveal}(\mathsf{tx})$ is true. After its release, the staker s_i can prove to SGX that its input is correct using the feed_revealed function. If the information provided is correct, then this information can be used to add to other stakers reveal and not invalidate the revealed transaction. Thus, such transactions will get marked as verified, and when all transactions are either verified or not committed to by the staker for the transaction sequence generated by Algorithm 1, then the collateral is returned to staker via refund (Line 48, Algorithm 3). $\square$

Lemma 5. *Given the staker s_i provides consistent input to Augmented Algorithm 1 and at the time of* reveal*, but the Algorithm 1 is aborted without returning tSeq, s_i will receive back the amount set as collateral.*

Proof. Even though no Algorithm 1 is complete, if the inputs to Algorithm 1 are the same as what the staker would release in order and reveal, the transaction sequence that follows would contain some transactions that the staker commit information towards, and some transactions that it did not commit information towards. The proof for Lemma 4 still holds. $\square$

Lemma 6. *Given the staker s_i provides inconsistent input to Augmented Algorithm 1 and at the time of* reveal *for some transaction* tx*, and a transaction sequence tSeq is published such that* $\mathsf{tx} \in$ *tSeq, s_i will not receive back the amount set as collateral.*

Proof. In order to receive back the collateral, the staker needs to mark all transactions in the transaction sequence tSeq and prove that tSeq was the confirmed transaction sequence on-chain. If the staker inputs tSeq in the update_MPC_block, then she would be required to mark the transaction. There exists two ways of marking a transaction - to not have committed to the information, and to show that the transaction's reveal corresponds to the input. If tx is not committed to, then the input to Algorithm 1 cannot contain tx (since otherwise the signature check would fail), and thus information provided cannot be incorrect. Next if tx is committed to, then the only way to get it marked is to show that the reveal of the transaction corresponded to the committed information. Since the checkInfo(Line 29, Algorithm 3) would fail for tx due to inconsistent input into Augmented Algorithm 1 and at the time of reveal. Thus, the staker would not be able to call refund. □

Lemma 7. *Assume that no user can distinguish whether any two public keys belong to the same entity except itself; and the contract in Algorithm 3 is indistinguishable from a HTLC contract. Suppose there exists a sequence of transactions tSeq =* $\{\bar{tx}_1, \dots, \bar{tx}_\ell\} \in \mathcal{P}(md_1, \dots, md_\ell)$ *for some input stream* $((md_1, data_1), \dots, (md_\ell, data_\ell))$*. Moreover, let us assume that there exists a function* extract() *known to all stakers such that* $tSeq' \models \mathsf{extract}(\mathsf{tx}_1, \dots, \mathsf{tx}_\ell)$ *and* $tSeq' \in \mathcal{P}(md'_1, \dots, md'_{\ell'})$ *for some input stream* $((md'_1, data'_1), \dots, (md'_{\ell'}, data'_{\ell'}))$*. Then, no user u can prove whether the input stream was* $((md'_1, data'_1), \dots, (md'_{\ell'}, data'_{\ell'}))$ *or some set of stakers* $\mathcal{A}'' \subseteq \mathcal{A}'$ *(with* $u \notin \mathcal{A}''$*) deviated from the protocol by running the above SGX code and the contract (Algorithm 3) in addition to Algorithm 1, when the input stream was* $((md_1, data_1), \dots, (md_\ell, data_\ell))$*.*

Proof. The only difference between the above stated lemma and Lemma 2, is that there exists an online contract, publicly visible to everyone. Since the contract has been designed as a Hashed Time Lock Contract (HTLC) [13], it cannot be used in any proof of malice. Note that incentive compatibility issues known for HTLC do not play any part in this, since there does not exist a second player and the address after timeout is just a burn address (which can be made indistinguishable from regular address as well). Note an HTLC design can be created even on non-smart contract based chains like Bitcoin. □

Lemma 8. *If there exists an* extract *function known to stakers such that* $tSeq' \models \mathsf{extract}(\mathsf{tx}_1, \dots \mathsf{tx}_\ell)$*, and utility of tSeq′ is greater than utility of tSeq, then even relaxing cryptographic binding property to a rational binding property of the reveal to any transaction in tSeq, publishing tSeq is strictly dominated by publishing the transaction sequence tSeq′ run from the MPC Algorithm 1.*

Proof. The staker s_i that releases the transaction sequence would choose a transaction sequence such that it maximizes its utility. Any choice the staker s_i chooses for the transaction sequence tSeq″ would have to yield a higher utility than tSeq, since there exists atleast tSeq′ which can be achieved by running extract which from the lemma statement has a utility greater than tSeq. Further, we also know that from Lemma 2, that no negative reward strategy can be applied for following MPC Algorithm 1. Thus, releasing transaction sequence tSeq is strictly dominated by releasing transaction sequence tSeq′. □

E Sandwich Analysis

In a normal sandwich attack, during the frontrunning transaction, the attacker swaps Δa_X of token X for Δa_Y of token Y changing the liquidity in the pool as $r'_X = r_X + (1-f)\Delta a_X$ and $r'_Y = r_Y - \Delta a_Y$ respectively. The value of Δa_X is adjusted such that the following equality holds:

$$\Delta r'_Y = (1-s)\frac{(1-f)r_Y \Delta r_X}{r_X + (1-f)\Delta r_X} = \frac{(1-f)r'_Y \Delta r_X}{r'_X + (1-f)\Delta r_X}$$

Then after the victim's transaction executes, all the Δa_Y is converted to X with the backrunning transaction.

We borrow the following result for optimal sandwich attack from [25, Theorem 2], which states that

Given that slippage is small (i.e., slippage determines optimal sandwich), lets first define η, a placeholder variable:

$$\begin{aligned}\eta = (1-f)^2(1-s)(\Delta r_X^2 (1-f)^4(1-s) \\ + 2\Delta r_X (1-f)^2(2-f(1-s))r_X \\ + (4 - f(4-f(1-s)))r_X^2)\end{aligned}$$

Then, the optimal attack input to the frontrunning transaction is given by

$$\Delta a_X = \frac{\frac{\sqrt{\eta}}{1-s} - \Delta r_X (1-f)^3 - (2-f)(1-f)r_X}{2(1-f)^2} \tag{1}$$

For ease of analysis, we define another placeholder variable,

$$\Gamma = \frac{\sqrt{\eta}}{(1-s)(1-f)} - \Delta r_X (1-f)^2 + f r_X$$

$$\text{such that } (1-f)\Delta a_X = \frac{\Gamma}{2} - r_X$$

F AnimaguSwap Analysis

In this subsection, we will analyze AnimaguSwap detailed in Fig. 3. For simplicity, we analyze the case where the attackers are able to sandwich the transaction and chose to guess the flip bit as $b = 0$. In the simulation, we would also take a look at when the attackers try to sandwich the other direction, i.e., guess $b = 1$.

Algorithm 3 A contract to add rational binding in the attacking protocol

```
1: $(\mathcal{K}, \textsc{enc}, \textsc{dec})$: defines an asymmetric encryption scheme
2: $\mathcal{H}$: represent a cryptographic hash function
3: // TEE side
4: State
5: secret: A secret revealed when correct mpc_inp$_i$ is verified
6: sk: stores a Secret Key generated inside TEE
7: inp: stores the committed mpc_inp$_i$ value for $s_i$
8: curr_tSeq: stores the last on-chain transaction sequence (block)
9: mpc_tSeq: stores the transaction sequence published on-chain (for which MPC was supposed to happen)
10: block_hash: block hash for the predecessor of MPC block.
11: function KEYGEN
12:     $\mathrm{sk}, \mathrm{pk} \leftarrow \mathcal{K}()$
13:     return pk
14: function GENERATE_HASH(mpc_inp$_i$, _block_hash)
15:     secret = Random()
16:     inp = mpc_inp$_i$
17:     block_hash = _block_hash
18:     return $\sigma = \mathsf{sign}_{\mathsf{sk}}(\mathcal{H}(\mathrm{secret}), \mathrm{inp}, \mathrm{block\_num})$
19: function UPDATE_MPC_BLOCK(tSeq)
20:     Assert tSeq.predecessor = block_hash
21:     mpc_tSeq = tSeq
22:     for tx $\in$ mpc_tSeq do
23:         if tx.txid $\notin$ mpc_inp$_i$.state.txid then
24:             mark(tx, mpc_tSeq)        ▷ Mark adds a mark on tx in the variable mpc_tSeq
25: function UPDATE_BLOCK(tSeq)
26:     Verify tSeq is successor of curr_tSeq
27:     curr_tSeq = blk
28: function FEED_REVEALED(tx, B)
29:     Assert CheckInfo(inp, tx)
30:     Assert CheckMembership(tx, B)
31:     if tx.txid $\in$ mpc_tSeq then
32:         mark(tx, mpc_tSeq)
33: function GET_PREIMAGE
34:     Assert checkConfirmed(mpc_tSeq)
35:     for tx $\in$ mpc_tSeq do
36:         if existsMark(tx, mpc_tSeq) then
37:             return null
38:     return secret
39: // Contract side
40: State
41: amount_stored $\leftarrow$ 0: amount stored in the contract
42: hash $\leftarrow$ null: hash of a secret the staker who is committing receives after running generate_hash
43: committer $\leftarrow$ null: identity of the staker that commits to the information
44: function COMMIT(amount, hash)
45:     amount_stored $\leftarrow$ amount_stored + amount
46:     hash = hash
47:     committer = sender
48: function REFUND(secret)
49:     if $\mathcal{H}$(secret) = hash then
50:         send(amount_stored, committer)
51: function CONFISCATE(timeout)
52:     if current.time > timeout then
53:         burn(amount_stored)
```

After the frontrunning transaction that swaps $\Delta \mathrm{a}_X$ of X for $\Delta \mathrm{a}_Y$ of Y, the update reserve pool r'_X and r'_Y are given by

$$\mathrm{r}'_X = \mathrm{r}_X + (1-\mathrm{f})\Delta \mathrm{a}_X = \frac{\Gamma}{2}$$

$$\mathrm{r}'_Y = \frac{\mathrm{r}_X \mathrm{r}_Y}{\mathrm{r}'_X}; \quad \Delta \mathrm{a}_Y = \mathrm{r}_Y - \mathrm{r}'_Y$$

If a flip does not happen (i.e., the user sets the flip bit to be 0), then the profit is given by P_S^+.

If a flip happens, i.e., the committee guesses the wrong direction while generating attacking transaction sequence, then the committee will release a transaction that instead of increasing the price for the user, would decrease it.

The victim transaction executed in this case would be swap $-\Delta \mathrm{r}_X$ of X for $\Delta \mathrm{r}'_Y$ of Y.

$$\Delta \mathrm{r}'_Y = \frac{\mathrm{r}'_Y(-(1-\mathrm{f})\Delta \mathrm{r}_X)}{\mathrm{r}'_X - (1-\mathrm{f})\Delta \mathrm{r}_X} = -\frac{4(1-\mathrm{f})\mathrm{r}_X \mathrm{r}_Y \Delta \mathrm{r}_X}{\Gamma(\Gamma - 2(1-\mathrm{f})\Delta \mathrm{r}_X)}$$

If no attack had happened, the expected swap would have been $-\Delta \mathrm{r}_X$ of X for $\Delta \mathrm{r}_Y^E$ of Y

$$\Delta \mathrm{r}_Y^E = -\frac{\mathrm{r}_Y(1-\mathrm{f})\Delta \mathrm{r}_X}{\mathrm{r}_X - (1-\mathrm{f})\Delta \mathrm{r}_X}$$

During backrunning (even though the transaction might fail due to slippage over allowance, we consider it to go through to see the loss incurred) the attacker would swap $\Delta \mathrm{a}_Y$ of Y for $\Delta \mathrm{a}'_X$ of X. Thus, the output would be $\Delta \mathrm{a}'_X = \frac{\Delta \mathrm{a}_Y(\mathrm{r}'_X - \Delta \mathrm{r}_X)}{(\mathrm{r}'_Y - \Delta \mathrm{r}'_Y + \Delta \mathrm{a}_Y)}$. The loss for the attacker in this case would be given by $\Delta \mathrm{a}'_X - \Delta \mathrm{a}_X$.

The profit for the flipper (which would be the excess tokens that user keeps) is given by $\Delta \mathrm{r}_Y^E - \Delta \mathrm{r}'_Y$.

Now, there also exists an option in which the attacker guesses that the user set the b to 1, for which a similar analysis would follow.

G Example Swap Scenario - Details of Execution

Table 2

Case	Transaction Type	Transaction Input	Transaction Output	SUSHI Reserve	WETH Reserve
Normal Victim Normal Attacker	Victim Expected ETFT	8592 SUSHI (+)	3.6551	173059.6482	70.1767
	Frontrun ETFT	842.4095 SUSHI (+)	0.3751	165310.0578	73.4567
	Victim ETFT	8592 SUSHI(+)	3.6189	173902.0578	69.8378
	Backrun ETFT	0.3751 WETH(+)	926.3164	172975.7413	70.2129
Flip Victim Normal Attacker	Victim Expected TFET	8592 SUSHI (-)	4.0819	155875.6482	77.9138
	Frontrun ETFT	842.4095 SUSHI (+)	0.3751	165310.0578	73.4567
	Victim TFET	8592 SUSHI (-)	4.0393	156718.0578	77.4961
	Backrun ETFT	0.3751 WETH (+)	752.6848	155965.373	77.8712
Flip Victim Flip Attacker	Victim Expected TFET	8592 SUSHI (-)	4.0819	155875.6482	77.9138
	Frontrun ETFT	0.2503 WETH(+)	554.0884	163913.5598	74.0822
	Victim TFET	8592 SUSHI(-)	4.1103	155321.5598	78.1926
	Backrun ETFT	554.0884 SUSHI(+)	0.2771	155875.6482	77.9155
Normal Victim Flip Attacker	Victim Expected ETFT	8592 SUSHI(+)	3.6551	173059.6482	70.1767
	Frontrun ETFT	0.2503 WETH(+)	554.0884	163913.5598	74.0822
	Victim ETFT	8592 SUSHI(+)	3.6793	172505.5598	70.4029
	Backrun ETFT	554.0884 SUSHI(+)	0.2247	173059.6482	70.1782

Table 2: An example swap where the Attacker sees: SwapExactTokenForToken 8592 SUSHI for WETH, when initial reserve of SUSHI is 164467.6483, WETH is 73.8319, with acceptable slippage of 1% (ETFT stands for SwapExactTokenForToken, TFET stands for SwapTokenForExactToken).