Data Independent Order Policy Enforcement: Limitations and Solutions

Sarisht Wadhwa
sarisht.wadhwa@duke.edu
Duke University
NC, USA

Luca Zanolini
luca.zanolini@ethereum.org
Ethereum Foundation
USA

Aditya Asgaonkar
aditya.asgaonkar@ethereum.org
Ethereum Foundation
USA

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

Chengrui Fang
Chengrui_Fang@zju.edu.cn
Zhejiang University
China

Fan Zhang
f.zhang@yale.edu
Yale University
USA

Kartik Nayak
kartik@cs.duke.edu
Duke University
USA

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) interface that is secure under rationality. We report on a prototype implementation of AnimaguSwap and performance evaluation results demonstrating its practicality.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**.

[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.

KEYWORDS

Blockchain, MEV, Crytoeconomics

ACM Reference Format:
Sarisht Wadhwa, Luca Zanolini, Aditya Asgaonkar, Francesco D'Amato, Chengrui Fang, Fan Zhang, and Kartik Nayak. 2024. Data Independent Order Policy Enforcement: Limitations and Solutions. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/3658644.3670367

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) [17]. An estimated $1.2B of MEV have been extracted as of the time of writing.[3]

To protect users from order manipulation attacks, an extensively explored direction [27, 26, 12, 8, 13, 28, 35, 4] is to design protocols that enforce certain "fair" transaction ordering policy. A popular approach is *data-independent* ordering, which 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 [27, 26, 12] order user transactions based on the *time* they are received by a *committee* of parties. Content-oblivious ordering (e.g., [8, 28, 35, 4]) guarantees that user transactions are hidden from the committee who orders them, e.g., through encryption, until after an ordering has been

[2]https://defillama.com
[3]https://explore.flashbots.net/

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., [26] assumes more than three-fourths 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 running the protocol may deviate from the protocols arbitrarily if doing so leads to a higher utility.

In this paper, we systematically investigate 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 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 [27, 26, 12, 8, 13, 28, 35, 4], 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).

The situation is a bit trickier for content-oblivious ordering, as collusion *might be accountable*. For example, in schemes where user transactions are encrypted using threshold encryption (e.g., [13]), enough parties can reconstruct the decryption key if they collude. However, this way of colluding may be accountable since the decryption key itself could serve as 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 first 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). We emphasize that the cost to collude between parties is very low since today's blockchain landscape is not so decentralized, and a few of the pools interacting with each other are all required to attack and collude. Further, once parties collude, they can profit for a longer duration, which decreases the amortized cost.

An order policy enforcement (OPE) framework and impossibilities. To prove this claim, we first present a generic framework in Section 4.1 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 4.2 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 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.

Introducing 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 $b \in \{0, 1\}$ and depending on this bit, creates a transaction that is either the one that the user intended (tx) or a related but different transaction ($\overline{\text{tx}}$), 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 [37, 24], 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 $\bar{b}$, 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 crafting $\overline{\text{tx}}$, which ensures that no stable collusion can be formed amongst the stakers.

In Section 5.1, we present AnimaguSwap, an Automated Market Makers (AMM) decentralized exchange that uses rationally binding commitments to defeat sandwich attacks assuming buying and selling a token is equally likely. In our protocol, if user transaction tx sells a certain asset, then $\overline{\text{tx}}$ 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.

In Section 5.2, we provide a game theoretic analysis of AnimaguSwap and show that following the protocol specification is dominant for all the involved parties. To evaluate practicality, in Section 5.3, we implement a base AnimaguSwap as a smart contract, and show that the key overhead in terms of the amount of gas is 1.3x compared to a typical insecure trade today. Moreover, since this cost does not depend on the number of stakers or the value of the transaction, this is already quite practical for high-value transactions. We then extend the result in Section 5.4 to consider scenarios where the buying and selling of a token may not be equally likely. We further enhance the security of the game to cover repeated games for a more practical solution, by ensuring the unlinkability between different games.

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 an impossibility proof showing that a wide range of OPE protocols cannot be secure when all parties are rational
- We propose the notion of rationally binding commitments as a practical way to circumvent the impossibility. We present the first AMM interface construction AnimaguSwap that can achieve data-independent ordering of user transactions in the presence of rational parties. We analyze the efficacy of AnimaguSwap by a game-theoretic proof in the presence of rational parties.
- We implement AnimaguSwap using a smart contract and show that the overhead of security is about 1.3x in gas cost compared to vanilla UniSwap; this will be practical for high-value transactions.

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 [17]. Below is a non-exhaustive list of protocols that are covered by the framework (Section 4.1) and the impossibility theorem (Theorem 1).

The first category of protocols is fair ordering. Kelkar *et al.* [27] 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 [25]. Subsequently, Kelkar *et al.* [26] devised Themis, a (permissioned) consensus protocol that, along the same lines as [27], achieves fair transaction ordering while preventing a liveness issue in Aequitas. Cachin *et al.* [12] 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., [13, 4, 35, 8]) 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 [28] efficiently integrates threshold encryption and secret sharing to a DAG-based BFT protocol. Shutter, Osmosis, and Sikka [13, 4, 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 [19, 29]. MEV auctions rely on the rationality of bidders (or builders) to maximize MEV extraction. Our solution (in Section 5) aims to achieve a different goal of reducing MEV.

Heimbach *et al.* [23] 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 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 [5] 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 ordering under rationality, though it suggested a TEE-based content-oblivious ordering protocol.

Lower bounds on MEV mitigation. Ferreira *et al.* [40] 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 [40]), 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. [40] 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). Moreover, [40] 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) [11], which our solution (Section 5.1) avoids.

3 MODEL AND PROBLEM STATEMENT

Throughout this paper, we consider *data-independent transaction ordering protocols* run by a set of N parties called *stakers* $\mathcal{S} = \{s_1, \ldots, s_N\}$. Such protocols process transactions submitted by *users* and output an ordered list of received transactions while ensuring that the ordering is independent of the transaction content. Examples include fair ordering based on receive order [27, 26, 12] and content-oblivious ordering (e.g., [8, 28, 35, 4]). The purpose of data-independent ordering is to prevent order manipulation attacks such as frontrunning attacks, sandwich attacks, etc. We refer readers to [41] for a survey of such attacks.

We assume all users including stakers to be *rational*, i.e., they act to maximize their utility function. To keep things simple, we assume that this utility function is the amount of monetary profit (in the number of *tokens*) that the party 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*.

Adversary model. Stakers are adversarial and they may deviate from the protocol arbitrarily if doing so increases their utility (after counting the penalty if any). Their goal is to tamper with the ordering process so that transactions are ordered to their advantage. For example, in receive order-based fair ordering protocols, stakers may collude and order a later transaction before an earlier one to facilitate a frontrunning attack; in content-oblivious ordering protocols, stakers may collude to decrypt user transactions and profit from the information thereof.

Problem statement. We ask two questions: First, existing data-independent ordering protocols are insecure under the above rationality assumption. Is this limitation fundamental or can it be mitigated? We answer this question negatively with an impossibility result. Second, given the impossibility, what relaxation of the problem can we make to obtain a practical data-independent ordering protocol under the above rationality assumption?

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 IMPOSSIBILITY OF OPE UNDER RATIONALITY

To study the common features of data-independent order policy enforcement (OPE) protocols [12, 26, 27, 28, 8, 13, 35, 4], we first present an abstract framework to capture the essence of aforementioned protocols with four sub-protocols (submit, process, order, reveal) and two predicates ShouldRelease and ShouldReveal. To aid understanding, we show how existing schemes can be mapped to our framework.

4.1 Framework for Order Policy Enforcement

Parties, transactions, and ordering policies. Our framework is run by *users*, who submit transactions, and a set of *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., on layer 2).

Definition 1 (Data and Metadata). *A transaction* tx_i *can be considered to consist of two parts – metadata* md_i *and data* $data_i$*. Metadata is defined as the part of a transaction not given to the application (i.e., a smart contract) for execution. Data is defined as the part of a transaction that is required for application execution.*

Our framework defines a generic protocol to enforce a data-independent policy $\mathcal{P}$.

Definition 2 (Data-independent Policy). *A policy is defined as data-independent if it takes as input a set of metadata (one for each*

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)
       $(\mathsf{txid}; (\bot, \mathsf{out}_1, \dots, \mathsf{out}_N)) \leftarrow \mathsf{submit}(\mathsf{tx}, \mathsf{inp}_1, \dots, \mathsf{inp}_N)$
   where 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
       $(\mathsf{tSeq} = (\bar{\mathsf{tx}}_1, \dots, \bar{\mathsf{tx}}_\ell); (\mathsf{state}_1, \dots, \mathsf{state}_N)) \leftarrow \mathsf{order}(\mathsf{state}_1, \dots, \mathsf{state}_N)$
   where the order of $(\bar{\mathsf{tx}}_1, \dots, \bar{\mathsf{tx}}_\ell)$ is dependent only on $\mathsf{md}_1, \dots, \mathsf{md}_\ell$.
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
       $(\mathsf{tx}_k; (\mathsf{state}_1, \dots, \mathsf{state}_N)) \leftarrow \mathsf{reveal}(\bar{\mathsf{tx}}_k; (\mathsf{state}_1, \mathsf{spri}_1), \dots, (\mathsf{state}_N, \mathsf{spri}_N))$
```

Figure 1: A general framework that captures proposed ordering policy enforcement protocols [12, 26, 27, 28, 8, 13] using four protocols (submit, manipulate, order, reveal) and two predicates ShouldRelease, ShouldReveal.

transaction) and outputs one or more permutations of transactions associated with them, i.e., $\mathcal{P}(md_1, \dots, md_\ell) \subseteq \sigma(\ell)$, where $\sigma(\ell)$ is the set of all permutations of $(\bar{\mathsf{tx}}_1, \dots, \bar{tx}_\ell)$.

Generally, each staker may have some different metadata for a given transaction, thus $\mathsf{md}_i = (\mathsf{md}_i^1, \dots, \mathsf{md}_i^N)$ represents the metadata for transaction tx_i across all N stakers.

The framework. As shown in Fig. 1, the framework for order policy enforcement consists of four sub-protocols. These subprotocols 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 param = (spri, 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,

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

, 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 $\bot$.
Users are ephemeral, i.e., they may go offline after running submit, a usability feature enjoyed by most real-world systems[12, 26, 27, 28, 8, 13]. Consequently, $(\mathsf{txid}; (\bot, \mathsf{out}_1, \dots, \mathsf{out}_N))$ 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 5.
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 $\mathsf{tSeq} = (\bar{\mathsf{tx}}_1, \dots, \bar{\mathsf{tx}}_\ell)$

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

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 $\bot$. 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 submit(tx, ·) 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 reveal() takes as input the output of the function 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 [32] or witness encryption [20]. 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[20]).

Examples. In Appendix A, we show that our framework can capture OPE protocols based on DKG [13, 8], secret-sharing [28], as well as fair ordering protocols [26, 12, 27].

4.2 Delineating Impossibility Conditions for Data Independent Ordering

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 $tSeq = \{\bar{tx}_1, \ldots, \bar{tx}_\ell\} \in \mathcal{P}(md_1, \ldots, md_\ell)$ with max utility for some input stream $((md_1, data_1), \ldots, (md_\ell, data_\ell))$. Moreover, let us assume that there exists a function* extract() *known to all stakers in $\mathcal{S}$ s.t. $tSeq' \models \mathsf{extract}(\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell) \in \mathcal{P}(md'_1, \ldots, md'_{\ell'})$ 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 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 [9, 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_i 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. We prove this formally in Appendix C. We show an example attack that follows the attack protocol described in Algorithm 1 in Appendix D.

5 OPE USING RATIONAL BINDING COMMITMENTS

In the impossibility result in the previous section, we assumed that given a sequence of transactions tSeq, the parties have access to an 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 [42], frontrunning [18, 31], arbitrage [18] etc. To make them work in the attack in Algorithm 1 (where tSeq is available but not in the clear), we can create an 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'.

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 $\bot$        ▷ $\mathsf{state}^a$ 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 $\bot$        ▷ $\mathsf{inp}^a$ 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 $\bot$        ▷ $\mathsf{spri}^a$ 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 ShouldRelease($s_i$) is true
5:     ($\mathsf{tSeq} = (\bar{\mathsf{tx}}_1, \dots, \bar{\mathsf{tx}}_\ell), \mathsf{state}^a) \leftarrow \mathsf{order}(\mathsf{state}^a)$        ▷ Validators in $\mathcal{A}$ order $\ell$ transactions
6:     for $j \in \{1, \dots, \ell\}$ 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:     $\mathsf{B} = (\mathsf{tx}_1, \dots, \mathsf{tx}_\ell)$
9:     VerifySigs(B)
10:    att_B $\leftarrow$ extract(B)        ▷ Get MEV-extracting transactions
11:    $\mathsf{state}' \leftarrow \bot$
12:    for att_txn $\in$ att_B do
13:        $(\mathsf{txid}; (\bot, \mathsf{out}_1, \dots, \mathsf{out}_N)) \leftarrow \mathsf{submit}(\mathsf{att\_txn}, \mathsf{inp}^a)$        ▷ 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:    $(\mathsf{tSeq}' = (\bar{\mathsf{tx}}'_1, \dots, \bar{\mathsf{tx}}'_{\ell'}); \mathsf{state}') \leftarrow \mathsf{order}(\mathsf{state}')$
17:    return $\mathsf{tSeq}', \mathsf{state}'_i$        ▷ Publish the block containing the MEV-extracting transaction
```

Importantly, for such an attack to work, indeed, the 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 x tokens of X for y tokens of Y with a slippage of s*" is to produce a sequence ($\mathsf{tx}^{\mathsf{attack}}_{\mathsf{BUY}}$, tx, $\mathsf{tx}^{\mathsf{attack}}_{\mathsf{SELL}}$) so that the first attacking transaction $\mathsf{tx}^{\mathsf{attack}}_{\mathsf{BUY}}$ reduces the supply of token X for tx making it pay a higher price, and $\mathsf{tx}^{\mathsf{attack}}_{\mathsf{SELL}}$ extracts the sandwiching profit. However, if the attackers are unaware whether tx was a buy or sell transaction, or if it may be reversed with some probability (i.e., tx became selling token X for Y), 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? In Section 5.1, we devise a strategy with *rationally binding commitments* by creating an information asymmetry (e.g., only one party knows whether it is a BUY or a SELL transaction) between a specific staker $\mathcal{F}$ (a flipper) and other stakers. In particular, the transaction can be modified after reveal has been invoked and $\mathcal{F}$ is responsible to *complete* the transaction. The asymmetry of information allows a rational $\mathcal{F}$ 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 $\mathcal{F}$ being rational, which is a reasonable assumption. In this world, the client needs only to monitor the chain and hold $\mathcal{F}$ accountable in case it observes $\mathcal{F}$ does not complete the transaction correctly.

We can also rely on users or TEEs held by stakers to withhold some information; this information is only revealed during the reveal phase. We discuss how to disincentivize attacks when this is possible in Appendix E. However, such a solution either requires the user to be online (which breaks the general ephemerality requirement) or needs additional assumptions, such as TEEs in the protocol. Since users do not have a stake, they typically tend to be ephemeral and this may cause liveness issues by not revealing their transactions.

5.1 AnimaguSwap

In this subsection, we describe a protocol design where some information is withheld from the attackers by a designated rational staker called flipper $\mathcal{F}$. This approach, as is, only works towards mitigating, and sometimes eliminating, sandwich attacks in constant product automated market makers (AMMs) like in Uniswap V2 [1]; though it can be easily extended to any constant function AMM. 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. 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.

5.1.1 Background. An Automated Market Maker (AMM) such as Uniswap [1], Balancer [6], and Curve[36], 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 $\mathsf{r}_X * \mathsf{r}_Y = k$ 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 $\Delta\mathsf{r}_X$ and receive an appropriate amount of the second asset $\Delta\mathsf{r}_Y$ in return. Each transaction to the AMM is charged an additional fee, which we represent by f (e.g., $\mathsf{f} = 0.3\%$ is a common value in practice). The constraint becomes $(\mathsf{r}_X + (1-\mathsf{f})\Delta\mathsf{r}_X) * (\mathsf{r}_Y - \Delta\mathsf{r}_Y) = k$. Such a trade is

SwapTokensForExactTokens in the Uniswap implementation and we represent it by Buy. Post the trade, the liquidity available would be $(\mathrm{r}_X + \Delta\mathrm{r}_X)$ and $(\mathrm{r}_Y - \Delta\mathrm{r}_Y)$ respectively (independent of the f).

A trade can be made with the fixed $\Delta\mathrm{r}_X$ amount in which to receive a fixed amount of first asset $\Delta\mathrm{r}_X$, the user deposits an appropriate amount of second asset $\Delta\mathrm{r}_Y$. Such a trade can be achieved by SwapExactTokensForTokens in Uniswap implementation and is referred to as Sell for the paper. The constraint for Sell is $(\mathrm{r}_X - (1-\mathrm{f})\Delta\mathrm{r}_X) * (\mathrm{r}_Y + \Delta\mathrm{r}_Y) = k$. Post the trade, the liquidity available would be $(\mathrm{r}_X - \Delta\mathrm{r}_X)$ and $(\mathrm{r}_Y + \Delta\mathrm{r}_Y)$ respectively.

Thus, given the current state of an AMM with 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 $\Delta\mathrm{r}_Y$. 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".

5.1.2 Sandwich Attack on Constant Product AMM. While slippage upper bounds users' loss, an attacker can still profit from user loss up to what is permitted by slippage by mounting *sandwich attacks*. This can be done by executing a transaction, depositing X, and receiving Y before the user's transaction (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 F for a mathematical analysis of the optimal frontrunning and backrunning parameters.

5.1.3 AnimaguSwap specification. We now present a protocol that can either reduce 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 that the user is interested in (token Y in our example). 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 Fig. 11 (Appendix I). It generally follows the structure of the framework in Figure 1 except for a couple of aspects that we will describe later. Recall that $\mathcal{F}$ refers to the flipper, a designated staker who would withhold the information from other stakers.

Transaction generation. Suppose the user intends to perform a trade from X to Y. This intent can be fulfilled in two ways: a "buy" transaction $\mathrm{tx}_{\mathrm{Buy}}$ that buys Y or, equivalently, a "sell" transaction $\mathrm{tx}_{\mathrm{Sell}}$ that sells X. With properly adjusted parameters, these two transactions have the same execution outcome. Specifically, we write $\mathrm{tx}_{\mathrm{Sell}} = \mathrm{Sell}(X, Y, \Delta\mathrm{r}_X, \Delta\mathrm{r}_Y, \mathrm{s}, \mathrm{md})$, where $\Delta\mathrm{r}_X$ represents the number of tokens of X to be sold, in order to get maximum possible Y units, which is expected to be $\Delta\mathrm{r}_Y$. The transaction would only go through if the number of tokens received $> (1-\mathrm{s})\Delta\mathrm{r}_Y$. md represents any other metadata to be used by the transaction. Similarly, $\mathrm{tx}_{\mathrm{Buy}} = \mathrm{Buy}(Y, X, \Delta\mathrm{r}_Y, \Delta\mathrm{r}_X, \mathrm{s}, \mathrm{md})$. In our notation, the first parameter is ($\Delta\mathrm{r}_X$ in case of Sell, and $\Delta\mathrm{r}_Y$ in case of Buy) is "exact" whereas the second parameter is determined by the first parameter and s.

The user first generates a random bit b to determine which transaction to use to fulfill its intended trade (note that the user is indifferent). Without loss of generality, we require that the user chooses the "buy" transaction $\mathrm{tx}_{\mathrm{Buy}}$ if and only if $\mathrm{b} = 0$. In the transaction metadata for tx_b, a hash of $v||w$ is included, where v and w are randomly generated numbers. This would be later used to allow slashing.

The key trick in AnimaguSwap is that the same coin decides if the user will "flip" the chosen transaction again. By flipping, we mean changing the polarity of the trade from selling asset X to buying asset X and vice versa, thereby creating a flipped transaction that is the opposite of the user's intent. Specifically, we require that the user flips the chosen transaction if and only if $\mathrm{b} = 1$. We denote the transaction after the optional flipping as tx_b.

Following the same example where the user intends to trade from X to Y. If $\mathrm{b} = 0$, the user will choose $\mathrm{tx}_{\mathrm{Buy}}$ and does not flip, i.e., $\mathrm{tx}_\mathrm{b} = \mathrm{tx}_{\mathrm{Buy}}$. If $\mathrm{b} = 1$, the user will choose $\mathrm{tx}_{\mathrm{Sell}}$ and flips, i.e., $\mathrm{tx}_\mathrm{b} = \mathrm{Buy}(X, Y, \Delta\mathrm{r}_X, \Delta\mathrm{r}_Y, \mathrm{s}, \mathrm{md})$. Note that in this case $\mathrm{tx}_\mathrm{b} \neq \mathrm{tx}_{\mathrm{Buy}}$. Also, the committee always receives a "buy" transaction from the user, but the true intent is hidden in the flip bit. As we will detail in the next step, the user will submit tx_b to the committee and the flip bit b to a different staker called the flipper $\mathcal{F}$.

The second key trick is to disincentivize the flipper from revealing the flip bit (b), by having the user create another transaction $\mathrm{tx}_{\mathcal{F}}$ which pays the flipper $\mathcal{F}$ some amount of tokens if stakers attempt to sandwich tx_b but the direction of the sandwiched transaction is opposite. In particular, following the same example, if $\mathrm{b} = 1$ and the committee creates a sandwich assuming $\mathrm{b} = 0$, the user will earn $\Delta\mathrm{r}'_Y > \Delta\mathrm{r}_Y$. It can then pay the flipper $\Delta\mathrm{r}'_Y - \Delta\mathrm{r}_Y$ without decreasing its utility from a no-attack scenario. Similarly, if $\mathrm{b} = 0$ and the committee assumes $\mathrm{b} = 1$, then the user would swap $\Delta\mathrm{r}'_Y < \Delta\mathrm{r}_Y$ and pay the flipper $\Delta\mathrm{r}_Y - \Delta\mathrm{r}'_Y$. To represent it mathematically, the user pays the flipper $\mathrm{b}(\Delta\mathrm{r}'_Y - \Delta\mathrm{r}_Y) + (1-\mathrm{b})(\Delta\mathrm{r}_Y - \Delta\mathrm{r}'_Y)$. Observe that obtaining $\Delta\mathrm{r}_Y$ is what the user expected; paying the remaining amount incentivizes $\mathcal{F}$. 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 forward a similar message to the stakers, and if this bit is incorrect the stakers would be able to slash the flipper. In order to safeguard against that, the user sends a random value v as an unsigned message to the flipper. It is crucial to ensure the deniability of the message sent by the user while maintaining the integrity of the message i.e., the message sent to the flipper could have been generated by the flipper itself. To ensure this, the user sends $m = (\mathsf{pk_u}||\mathsf{b}||v||\mathsf{txid})$ to the flipper encrypted under $\mathsf{pk_{flipper}}$ using a hybrid public key encryption scheme (e.g., [7]). The message sent above could only be generated by a party who knows the correct random number v and the transaction ID txid. This ensures that no party except for the user and flipper (the two parties that know the content of the message) could have generated the message.

When returning the signed message to the user committing to b, it also includes v in the commitment. The user then generates another random number w, and uses hash(v||w) in the transaction metadata. This ensures that only the user or a party with w can slash the flipper using the signed message the flipper sent, and thus, the flipper is free to sign any message it wants without risk of getting slashed.

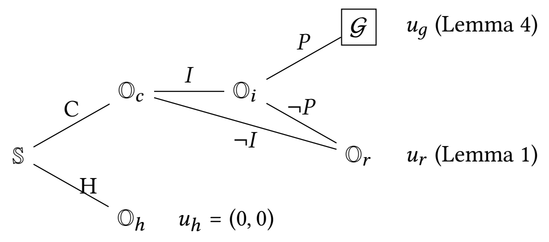
Once both these steps succeed, the user secret-shares the (potentially flipped) transaction with the remaining stakers.

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, $\mathcal{F}$ reveals the bit b too so that the correct transaction is revealed.

Pessimistic slashing. In case the flipper reveals bit $\tilde{\mathsf{b}}$ instead of b, then the user uses the signed message $(\mathsf{b}, \mathsf{txid}, v)_{\sigma(\mathsf{pk}_{\mathcal{F}})}$ in addition to v and w to slash the flipper. The slashing rule gives us the following guarantees:

- Correctness: The user can only slash the flipper in case an incorrect bit is revealed, as slashing requires the user to show a signed flip bit different from what the flipper revealed. Correctness follows from the unforgeability of digital signatures.
- Soundness: If the flipper releases an incorrect bit, then the user can slash the flipper. Since the user has the signed message which contains the correct bit b, it acts as a commitment by the flipper and since both v and w are known to the user, the signature can be used to slash the flipper by showing the authenticity of v (revealing v and w, and verifying it against $hash(v||w)$).
- Non-transferability: The flipper cannot convince any party (other than the user) that $(\mathsf{b}, \mathsf{txid}, v)_{\sigma(\mathsf{pk}_{\mathcal{F}})}$ can be used to slash the flipper. Note that this message can slash the flipper only if v is committed to by the metadata $md = hash(v\|w)$. Since w is private to the user, md is a perfectly hiding commitment to v, so no party can verify that flipper's claimed v is committed to by md, following the definition of hiding.

Observations. 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 user. 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. The user 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.



(a) Sequential Game

$\mathcal{F}$ \ C	Accept information	Anticipate Betray
Co-operate	u_s (Lemma 2)	u_{rs} (Lemma 3)
Betray	u_{rs} (Lemma 3)	u_s (Lemma 2)

(b) $\mathcal{G}$- Simultaneous Game

Figure 2: Game Tree for actions taken in AnimaguSwap. Consists of a sequential game in which each of $\mathcal{F}$ and C decide to participate or not and if both decide to participate, there exists a simultaneous game to decrypt the transaction correctly or incorrectly.

5.2 AnimaguSwap Analysis

In this subsection, we will analyze AnimaguSwap detailed in Section 5.1.3. Our goal is to show that following protocol specifications is the dominant strategy for all the parties involved. For ease of analysis, we assume that the committee is colluding (e.g., through a collusion protocol such as Algorithm 1), and hence, treat the committee as a single party. We start the game after the user sends the flip bit to the flipper $\mathcal{F}$, and the transaction is secret shared with the committee C. For the analysis, C reconstructs the secret transaction sent to it. Also, the analysis would follow a single-shot game (i.e., the flipper and the committee are not repeated). In more practical scenarios, the game would be a multi-shot game. The reduction from a multi-shot game to a single-shot game is shown in Appendix G.

[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 [13], Ferveo [8], or Fino [28].

5.2.1 *Game Setup.* We analyze the game for a single AnimaguSwap transaction. The game underlying AnimaguSwap consists of two players - the flipper ($\mathcal{F}$), who receives one bit of information from the user on whether or not to flip the direction of the trade, and the committee (C), which receives the transaction.

Definition 3 (AnimaguSwap Game). *We define the game as a tuple (N, A, O, μ, u), where $N = \{\mathcal{F}, C\}$, $A = \{A_{\mathcal{F}}, A_C\}$ is the set of actions available to $\mathcal{F}$ and C respectively, $O = \{\mathbb{O}_h, \mathbb{O}_r, \mathbb{O}_s, \mathbb{O}_{rs}\}$[5] is the set of outcomes from the game, μ is the function that maps the set of actions to the outcome, and $u = \{u_h, u_r, u_s, u_{rs}\}$ is utility corresponding to outcomes.*

The game states achievable through the game are $\mathbb{S} = \{\mathbb{O}_c, \mathbb{O}_i, \mathbb{O}_h, \mathbb{O}_r, \mathbb{O}_s, \mathbb{O}_{rs}\}$ The action space for the game is defined as A_C = {H (Honest), C (Collude), I (Invite $\mathcal{F}$), AI (Accept Information), AB (Anticipate Betray)} and $A_{\mathcal{F}}$ ={P (Participate), Co (Cooperate), B (Betray)}. The game tree Fig. 2 helps to understand action space better and also designs the function μ.

We assume that both players are rational, and have perfect information on strategy used by the other player, a concept used to find a Nash equilibrium of the game. We will first show such an equilibrium of strategy in a simultaneous game (Fig. 2(b)), and then plug in the utility for the equilibrium strategy in the sequential game (Fig. 2(a)), to use iterative elimination of dominated strategies to find the best strategy across both games to earn the highest utility.

In this model, a player's utility is defined by the net number of tokens gained or lost in the game. Further, the action set for the game is limited, and any actions such as being involved in binding side contracts are out of the scope of this analysis. We will discuss the case with binding side contracts in Appendix J. There are two ways to reach outcomes $\mathbb{O}_{rs}$ and $\mathbb{O}_s$, but the players' utility is only a function of the state, not the actions leading to the state (by definition of utility function). Specifically, in case the game reaches $\mathbb{O}_s$, i.e., a successful sandwich attack due to either {Co, AI} actions or {B, AB}, $\mathcal{F}$ and C get a utility independent of actions taken to reach the state; In case the game reaches $\mathbb{O}_{rs}$, i.e., unsuccessful sandwich attack due to either {Co, AB} actions or {B, AI}, $\mathcal{F}$ does not lose any utility due to the actions taken. As an example of utility sharing, after the action I a conditional bribe can be set to $\mathcal{F}$, which would only go through if the sandwich is successful.

From Section 5.1.1, without any attacker transaction, if the user's transaction was Sell then it would have followed

$$(\mathrm{r}_X + (1-\mathrm{f})\Delta\mathrm{r}_X)(\mathrm{r}_Y - \Delta\mathrm{r}_Y^S) = \mathrm{r}_X\mathrm{r}_Y \tag{1}$$

Without any attacker transaction, if the user's transaction was Buy then it would have followed

$$(\mathrm{r}_X - (1-\mathrm{f})\Delta\mathrm{r}_X)(\mathrm{r}_Y + \Delta\mathrm{r}_Y^B) = \mathrm{r}_X\mathrm{r}_Y \tag{2}$$

5.2.2 *Analysis.* Let us represent direction as a random variable chosen uniformly by the user from {Buy, Sell}. Without loss in generality, we can always represent the frontrunning transaction as a Sell transaction. In the case of Buy, the fee would be charged from the other token, but the essence of the proof would remain the same. First, the frontrunning transaction would follow the constant product invariant.

$$(\mathrm{r}_X + (1-\mathrm{f})\Delta\mathrm{a}_X)(\mathrm{r}_Y - \Delta\mathrm{a}_Y) = \mathrm{r}_X\mathrm{r}_Y \tag{3}$$

Now, after the frontrunning transaction, the victim transaction would follow. The transaction here could be a Sell transaction or Buy transaction, following the same set of parameters as C's frontrunning transaction. If the user's transaction is Sell, then it would follow the constant product invariant.

$$\begin{aligned}(\mathrm{r}_X + \Delta\mathrm{a}_X + (1-\mathrm{f})\Delta\mathrm{r}_X)(\mathrm{r}_Y - \Delta\mathrm{a}_Y - \Delta\mathrm{r}_Y^+)\\ = (\mathrm{r}_X + \Delta\mathrm{a}_X)(\mathrm{r}_Y - \Delta\mathrm{a}_Y)\end{aligned} \tag{4}$$

Also, the user's transaction would only be executed if

$$\Delta\mathrm{r}_Y^+ > (1-\mathrm{s})\Delta\mathrm{r}_Y^S \tag{5}$$

The backrunning transaction would follow the constant product invariant with updated liquidity pools.

$$\begin{aligned}(\mathrm{r}_X + \Delta\mathrm{a}_X + \Delta\mathrm{r}_X - \Delta\mathrm{a}_X^+)(\mathrm{r}_Y - \Delta\mathrm{a}_Y - \Delta\mathrm{r}_Y^+ + (1-\mathrm{f})\Delta\mathrm{a}_Y)\\ = (\mathrm{r}_X + \Delta\mathrm{a}_X + \Delta\mathrm{r}_X)(\mathrm{r}_Y - \Delta\mathrm{a}_Y - \Delta\mathrm{r}_Y^+)\end{aligned} \tag{6}$$

The profit would be given by

$$p^+ = \Delta\mathrm{a}_X^+ - \Delta\mathrm{a}_X \tag{7}$$

Since the sandwich is successful, $\Delta\mathrm{r}_Y^+ < \Delta\mathrm{r}_Y^S$. If the transaction is Buy, then it would follow Eqs. (8) to (11),

$$\begin{aligned}(\mathrm{r}_X + \Delta\mathrm{a}_X - (1-\mathrm{f})\Delta\mathrm{r}_X)(\mathrm{r}_Y - \Delta\mathrm{a}_Y + \Delta\mathrm{r}_Y^-)\\ = (\mathrm{r}_X + \Delta\mathrm{a}_X)(\mathrm{r}_Y - \Delta\mathrm{a}_Y)\end{aligned} \tag{8}$$

$$\Delta\mathrm{r}_Y^- > (1-\mathrm{s})\Delta\mathrm{r}_Y^B \tag{9}$$

$$\begin{aligned}(\mathrm{r}_X + \Delta\mathrm{a}_X - \Delta\mathrm{r}_X - \Delta\mathrm{a}_X^-)(\mathrm{r}_Y - \Delta\mathrm{a}_Y + \Delta\mathrm{r}_Y^- + (1-\mathrm{f})\Delta\mathrm{a}_Y)\\ = (\mathrm{r}_X + \Delta\mathrm{a}_X - \Delta\mathrm{r}_X)(\mathrm{r}_Y - \Delta\mathrm{a}_Y + \Delta\mathrm{r}_Y^-)\end{aligned} \tag{10}$$

$$p^- = \Delta\mathrm{a}_X^- - \Delta\mathrm{a}_X \tag{11}$$

Since the sandwich is unsuccessful, $\Delta\mathrm{r}_Y^- < \Delta\mathrm{r}_Y^B$

Lemma 1. *If the transaction's direction is uniformly distributed between {Buy, Sell}, and if C takes the action to cooperate (C), and either takes action to not invite $\mathcal{F}$ ($\neg I$), or after taking action to invite $\mathcal{F}$ (I), $\mathcal{F}$ does not participate ($\neg P$), to reach output state $\mathbb{O}_r$, then independent of the direction of trade C chooses, the utility of $\mathcal{F}$ and C ($u_r = u_r(\mathcal{F}), u_r(C)$) is given by $(\frac{1}{2}(\Delta r_Y^B - \Delta r_Y^-), \frac{1}{2}(p^+ + p^-))$.*

Proof. If C does not have information about the direction of the trade, then it can assume a direction among {Buy, Sell}. Without loss of generality, we represent the committee's frontrunning transaction in the form of a Sell($\mathrm{r}_X, \mathrm{r}_Y, \Delta\mathrm{a}_X, \Delta\mathrm{a}_Y$, s, md).

Since the direction of transaction is chosen at random from {Buy, Sell}, Eq. (7) and Eq. (11) govern the profit with probability 0.5 each, and the expected utility would be given by

$$u_r(C) = \frac{p^+ + p^-}{2} \tag{12}$$

Next, the utility for $\mathcal{F}$ would be given from the AnimaguSwap protocol only in the case when C guesses the transaction direction incorrectly (and 0 in the other case).

[5] h stands for honest, r random, s sandwich, and rs reverse sandwich.

$$u_r(\mathcal{F}) = \frac{1}{2}(\Delta r_Y^B - \Delta r_Y^-) \tag{13}$$

□

Lemma 2. *For game $\mathcal{G}$, if $\mathcal{F}$ takes the action to co-operate (Co), and C accepts the information (AI), or $\mathcal{F}$ takes the action to betray (B), and C anticipates betrayal (AB), then the utility is given by $u_s = (u_s(C) = \varepsilon, u_s(\mathcal{F}) = p^+ - \varepsilon)$, where $0 < \varepsilon < p^+$.*

Proof. The proof for the lemma follows Eq. (7). The profit gained from choosing the correct direction to sandwich would be shared between $\mathcal{F}$ and C, regardless of the actions taken to reach the state. Thus, if C receives a utility of ε, then $\mathcal{F}$ receives a utility of $p^+ - \varepsilon$, $0 < \varepsilon < p^+$. $\mathcal{F}$ receives no utility directly from the AnimaguSwap protocol. □

Lemma 3. *For game $\mathcal{G}$, if $\mathcal{F}$ takes the action to co-operate (Co), but C anticipates betrayal (AB), or $\mathcal{F}$ takes the action to betray (B), but C accepts the information (AI), then the utility is given by $u_{rs} = (u_{rs}(C) = p^-, u_{rs}(\mathcal{F}) = \Delta r_Y^B - \Delta r_Y^-)$.*

Proof. Both sets of actions lead to a state where the committee inserts a frontrunning transaction with the incorrect direction. As stated in the setup, this would mean that $\mathcal{F}$ has no utility from the game itself, and C loses utility governed by p^-(Eq. (11)). However, in accordance with the AnimaguSwap protocol, $\mathcal{F}$ receives incentives from the protocol. This would be given by $\Delta r_Y^B - \Delta r_Y^-$. Thus $u_{rs} = (p^-, \Delta r_Y^B - \Delta r_Y^-)$. □

Lemma 4. *For game $\mathcal{G}$, if $\Delta r_Y^B - \Delta r_Y^- > p^+ - \varepsilon$ the Nash Equilibrium is governed by a mixed strategy for both $\mathcal{F}$ and C, with $\mathcal{F}$ betraying the committee with a probability of 0.5, and C anticipating betrayal with a probability of 0.5. The overall utility from game $\mathcal{G}$ is given by $\left(\frac{\varepsilon + p^-}{2}, \frac{p^+ - \varepsilon + \Delta r_Y^B - \Delta r_Y^-}{2}\right)$*

Proof. To prove that the above strategy is a Nash Equilibrium, we reveal the strategy of each player to the other player and see if the strategy changes. From $\mathcal{F}$'s perspective, if it knows that C anticipates betrayal with a probability of 0.5, then the expected utility from betraying is $\frac{(u_s(\mathcal{F}) + u_{rs}(\mathcal{F}))}{2}$, whereas the utility from cooperating is $\frac{(u_s(\mathcal{F}) + u_{rs}(\mathcal{F}))}{2}$. From lemmas 2 and 3, both of these are equal, and thus $\mathcal{F}$ does not have any additional utility from deviating from the strategy.

From C's perspective, if it knows that $\mathcal{F}$ betrays with a probability of 0.5, then the expected utility from anticipating betraying is $\frac{(u_s(C) + u_{rs}(C))}{2}$, whereas the utility from accepting the information is $\frac{(u_s(C) + u_{rs}(C))}{2}$. From lemmas 2 and 3, both of these are equal, and thus C does not have any additional utility from deviating from the strategy.

Thus, the given strategy is a Nash Equilibrium and by substituting the utilities from lemmas 2 and 3, the utility is given by $\left(\frac{\varepsilon + p^-}{2}, \frac{p^+ - \varepsilon + \Delta r_Y^B - \Delta r_Y^-}{2}\right)$. □

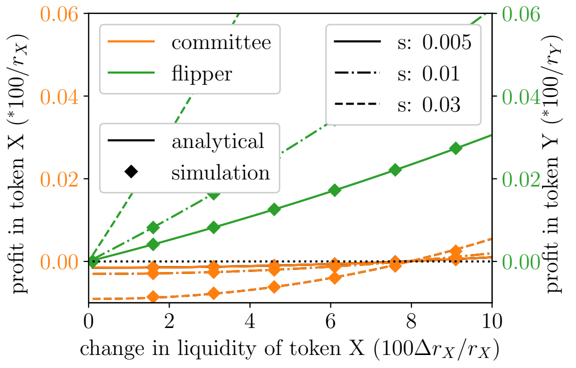


Figure 3: Gains of the committee and the flipper in a sandwich attack when using AnimaguSwap.

Lemma 5. *In the game state $\mathbb{O}_c$, if the transaction's direction is uniformly distributed between {Buy, Sell}, and if $\Delta r_Y^B - \Delta r_Y^- > p^+ - \varepsilon$, it is strictly dominant for C to not invite $\mathcal{F}$ ($\neg I$) over inviting $\mathcal{F}$ (I).*

Proof. From Lemma 1, the utility of C from random choice is $u_r(C) = \frac{p^+ + p^-}{2}$. From Lemma 4, the utility of C from game $\mathcal{G}$ is $\frac{\varepsilon + p^-}{2}$. Since $\varepsilon < p^+$, $\frac{p^+ + p^-}{2} > \frac{\varepsilon + p^-}{2}$. Thus, choosing the direction at random strictly dominates colluding with $\mathcal{F}$. □

Theorem 2. *If the transaction's direction is uniformly distributed between {Buy, Sell}, $\Delta r_Y^B - \Delta r_Y^- > p^+$ and $\Delta a_X^+ + \Delta a_X^- - 2\Delta a_X < 0$, it is dominant for all parties in AnimaguSwap to follow the specification in Section 5.1.3.*

Proof. If $\Delta r_Y^B - \Delta r_Y^- > p^+$, then $\Delta r_Y^B - \Delta r_Y^- > p^+ - \varepsilon$ since $\varepsilon > 0$. Thus, from Lemma 5, it is strictly dominant for C to choose the direction of trade arbitrarily over colluding with $\mathcal{F}$. If $\Delta a_X^+ + \Delta a_X^- - 2\Delta a_X < 0$, then the expected profit from attacking AnimaguSwap by arbitrarily choosing a direction of trade is < 0 by plugging in p^+ and p^- in Lemma 1. Since taking honest actions leads to 0 utility, the committee would choose to take honest actions in AnimaguSwap. □

The chart (Fig. 3) represents the variation of the maximum utility with the user's transaction input (Δr_X) relative to the liquidity of X available (r_X), where slippage is set to be 0.005, 0.01, and 0.03 respectively. If both $\mathcal{F}$ and C act honestly, they both receive no utility from the game. From the chart, we observe that $u_r(C) < 0$ until $\sim 7.9\%$ of the liquidity is traded. From this, we can conclude that C and consequently $\mathcal{F}$ would act honestly unless the traded amount $> 7.9\%$ of the liquidity. 7.9% value is roughly when $\Delta a_X^+ + \Delta a_X^- - 2\Delta a_X = 0$.

To validate the analysis when C chooses the direction of trade arbitrarily, we simulate the attack in an AMM modeled after Uniswap v2, and calculate the expected gains for the attacker and the flipper when the attacker decides to sandwich the transaction across multiple values of s set by the user. Figure 3 shows that the analytical and simulation results are consistent.

Numerical example. As a concrete example, Table 2 in Appendix H shows the loss/gain of an attack against a simulated user trade sent through AnimaguSwap to Uniswap V2.

Function	Gas cost	Function	Gas cost
commit	66275	revealFlipper	46069
revealStaker	239342	complain	34862

Table 1: Cost for a AnimaguSwap call. It takes 1564706 gas to deploy (one-time cost). Complain is not on critical path.

5.3 Base AnimaguSwap Evaluation

To evaluate the practicality of AnimaguSwap, we implemented the protocol (without the sub-routines for non-uniform distribution for {Buy, Sell} and repeated games) as a smart contract AS in 350 lines of Solidity. At a high level, AS is a middleware between users and an AMM, where users send commitments of transactions to AS following the AnimaguSwap protocol, stakers and flipper reveal user transactions, and finally, AS forwards the revealed transactions to the destination. In our prototype implementation, the destination is a fork of Uniswap V2 AMM. Specifically, the smart contract handles the following tasks:

- **Staking and slashing.** Stakers and the flipper deposit an appropriate amount of collateral to AS to join the system. When misbehavior is detected, users can submit evidence to AS to slash their stakes (c.f. Pessimistic Slashing Fig. 12).
- **Commit.** To commit to transaction tx, a user runs *Generate Transaction* and *Transaction Submission* specified in Section 5.1.3. After interacting with stakers and flippers, the user calls AS.commit with the hash of tx_b.
- **Reveal and execution.** The reveal process follows *Transaction Revealing* in Section 5.1.3. Specifically, the stakers reconstruct the secret shared transaction off-chain and one of the stakers calls AS.revealStaker with the reconstructed transaction tx_b. revealStaker verifies the correctness of tx_b against the commitment. Then, Flipper calls AS.revealFlipper with flip bit b. With both b and $\mathrm{tx_b}$, AS recovers the user's original intent and executes it. In our implementation, this triggers a call to Uniswap V2.

Off-chain parties (stakers and the flipper) are implemented in 400 lines of TypeScript. The code can be found at https://anonymous.4open.science/r/AnimaguSwap-D31F/.

Evaluation. The stakers' main task is to reconstruct user transactions from secret shares to open commitments and execution transactions. The flipper's task does not involve any costly computation. Compared to smart contract execution, off-chain computation is much more efficient (see, e.g., [8], for evaluation). The main performance metric therefore is the gas consumption of AS.

Table 1 shows the gas cost breakdown of executing a Uniswap trade from 1 wBTC to DAI, which costs 351k gas ($4.6 at the time of writing). To compare, a typical Uniswap V2 trade costs about 150k gas ($2). [6] The strong protection of AnimaguSwap thus incurs a 1.3x overhead. Note that since transactions are reconstructed off-chain, the gas cost does not increase with the number of stakers. Also, note that the cost does not increase with the value of the transaction. Therefore, while gas usage can be potentially reduced further, our preliminary implementation is already quite practical for high-value transactions.

[6]https://etherscan.io/gastracker

5.4 Non-uniform Distribution in {Buy, Sell}

In Theorem 2, we assumed for the result that there is an even spread between two trade directions, {Buy, Sell}. However, this assumption might not hold in real-world scenarios. For instance, during the LUNA crash of May 2022, a trade involving ETH and LUNA is more likely to be selling LUNA, than buying it. In such scenarios, AnimaguSwap as presented above does not work, as the attacker can *guess* the flipper's bit based on public information such as market sentiment. Another problematic scenario is when the user only owns one of the assets in her transaction, so the attacker can deduce that the transaction must be selling that asset.

Formally, for a given pair of assets $Pair = (X, Y)$, if the probability of trading X to Y in a random transaction is different from that of the reverse direction, we say this asset pair is *biased*, and we denote the probability of the more probable direction with $\mathrm{P\_DIR}_{Pair} \geq 0.5$. (The probability of the less probable direction is $1 - \mathrm{P\_DIR}_{Pair}$.) We call $\mathrm{P\_DIR}_{Pair}$ the *bias* of the pair for short. We omit the subscription when the asset pair is clear from context. Now we present an enhancement to AnimaguSwap that can protect biased asset pairs.

Our idea is to obfuscate the user transaction (using techniques to be presented shortly) so that an attacker mounting sandwich attacks based on guessed transaction information will equally likely fail or succeed. 'Success' here means the attacker manages to profit from the sandwich attack, while 'failure' means the attacker loses. As long as the amounts gained or lost in a 'success' and 'failure' remain the same as in the gains and loss in Section 5.2, all the lemma statements and the theorem statement would follow. We present the subroutines integrated into AnimaguSwap in Appendix I.

In order to get an equal probability of 'Success' and 'Failure', we take the following three steps: 1) remove any user dependency by creating a pool of users; 2) hide the asset being traded among multiple transactions in a way that no party can distinguish which asset is being traded, and 3) set the parameters so that the attacker can only lose utility from sandwiching incorrectly, but never gain any profit, if it chooses the wrong asset. With these three properties, we make it such that the attacker gains a profit when it gets everything correct, however, in multiple cases where it gets the trade direction incorrectly, the attacker loses. There also exist cases where the attacker neither wins nor loses (pays some transaction fee), but we assume the utility for such cases is 0.

To start with, we obfuscate the user identity by mixing it among a pool of decoy users, so that the attacker cannot gain information about the user's transaction from the user's on-chain presence (e.g., if this user doesn't possess the asset X, then the attacker can infer that the user cannot only be buying X.) To select decoy users, the user randomly chooses users who collectively own all the assets being traded (the real asset pair, and all the auxiliary asset pairs which we would introduce next). The selection is made such that given all the assets (including auxiliary assets, which will be introduced soon) involved in the trade, selecting any user out of the pool gives no extra information about the trade to the attacker over any other user in the pool. For example, let's say the user submits a trade between USDC and ETH. It has USDC, but not ETH. The user would create a pool of users (from the set of all blockchain users), such that one user has ETH but no other assets. This pool of users (in this case, 2 users are sufficient, but adding more users to have redundancy does not hurt) would be used to generate a ring signature [33], such that no party can distinguish which user amongst the pool of users created the transaction. The flipper escrows the true identity of the transaction creator and will be released in the same way as the flip bit.

However, in doing so, the flipper can release incorrect information about the user sending the transaction. i.e. it can create a transaction itself and generate a transaction, but since the identity is obscured, it releases the wrong identity. To prevent this, we need a way for a user to prove that the transaction is not its own. We do this by controlling the v variable used in metadata creation. It was introduced to hide the commitment of the commitment received from Flipper to the information it has. Instead of randomly generating it, the user uses $v = h(sk, txid)$, where sk is the secret key, and $txid$ is the block number. To any party without sk, it remains a completely random number, but any user can generate a zero-knowledge proof that it is not the user generating the transaction.

We also introduce the concept of auxiliary transactions. The idea behind the concept is to hide how much is being traded and which pair of assets is involved. Consider, for instance, the bias of the user's "real" transaction is $\text{P\_DIR}_{real} > 0.5$. To achieve an equal probability of 'Success' and 'Failure', the user chooses N_ASSET different asset pairs with a skew of the direction of trade similar to $\text{P\_DIR}_i > 0.5$ for each asset pair i, the value of which we will define shortly. Next, it creates N_ASSET auxiliary transactions with the newly chosen pairs, the same value as the original transaction, and chooses the direction of the trade with the same probability P_DIR_i. We refer to P_ASSET_{real} as the probability that amongst a set of assets, real is the actual asset pair, and P_ASSET_i for introduced auxiliary asset pair i, such that $\text{P\_ASSET}_{real} + \sum_i \text{P\_ASSET}_i = 1$. For example, let's say for the ETH-USDC pair, there exists a probability of 0.6 for traders to buy ETH from USDC. The user would find another pair (e.g., WBTC-Tether) such that the attacker cannot differentiate whether this transaction was an ETH-USDC pair or the WBTC-Tether pair. We will show how this other pair (WBTC-Tether) is found after the claim statement. The slippage for this transaction is such that at the current price, the trade would fail, however, at any price better than the current price, the transaction would succeed. This way, the attacker can only 'Fail' if it chooses to attack the auxiliary assets, and not 'Succeed' even when the direction guessed is correct.

Introducing new auxiliary transactions would require the token being traded to be available to the user. However, the user may not own auxiliary assets to make transactions in the first place. One workaround is to borrow auxiliary assets from a decentralized lending platform in an indistinguishable way (indistinguishable which asset is being borrowed) from the attacker. A naïve solution to this is to loan all possible assets involved in the trade, with the smallest union of assets not involved in the trade over the user pool. e.g. If assets A and B are the real traded assets, and C, D, E, and F are used as auxiliary assets, and in the user pool, all users have either asset G, H, or I, then a loan for A, B, C, D, E, F would be required keeping an asset G, H and I as collateral (i.e., a total of 18 loans). Most of these loans would fail since the user would not have the required collateral to obtain the loan. However, with AnimaguSwap, we have an advantage that at the time of execution, we have complete knowledge, and hiding the access is not important. Thus, if we can program AnimaguSwap contract to involve generating transactions for loans, we can optimize the process to not involve unnecessary loan transactions. In addition, the collateralized asset needs to be outside the set of assets involved in the trade (the main transaction and the auxiliary transaction). For example, in the ETH-USDC trade, if the user chooses the auxiliary pair as WBTC-Tether (Let's say WBTC is being bought more than Tether), then it would need to ensure that the auxiliary transaction is valid. If it does not have the asset being traded away (Tether), then the AnimaguSwap would issue a loan for Tether keeping SUSHI (A third asset not involved in the trade) as collateral. This is important for the user ambiguity constraint since if the user does not have an asset for collateral, it gives the attacker information that the user may be more likely to use the asset it has as the traded asset. If the user is using a loan from an external asset, all the assets involved are equally likely, however, in case an asset is used that is in the set of traded assets, then that asset is more likely to be a traded asset (real asset, or auxiliary traded asset).

With the above-described changes to the AnimaguSwap specification (Fig. 12), we can claim the following:

Claim 1. *Given a probability of the trade being $\text{P\_DIR}_{real} > 0.5$ in one direction, if there exists a set of asset pairs with probability $\text{P\_DIR}_i > 0.5$ of it being traded in a given direction, such that $\sum_i(\text{P\_ASSET}_i * \text{P\_DIR}_i) = 1 - 2\text{P\_ASSET}_{real} * \text{P\_DIR}_{real}$, where P_ASSET_i represents the probability of asset pair i to be the real asset pair amongst the set of asset pairs chosen for a pool of users, given that the user owns an asset not involved in the trade as collateral, then the subroutine described ensures that the probability of an attacker successfully gaining utility from sandwich is equal to probability of an attacker losing utility.*

Proof. The protocol creates the following scenarios when sandwiched attacked - 1) the attacker chooses both the asset and direction of trade correctly (probability = $\text{P\_ASSET}_{real} * \text{P\_DIR}_{real}$); 2) chooses the asset correctly, but the direction is inverted (probability = $\text{P\_ASSET}_{real} * (1 - \text{P\_DIR}_{real})$); 3) chooses the asset incorrectly but the direction is the same as the direction chosen for the dummy transaction (probability = $\sum_i(\text{P\_ASSET}_i * \text{P\_DIR}_i)$); and 4) chooses the asset as well as the direction of the dummy transaction incorrectly (probability = $\sum_i(\text{P\_ASSET}_i * (1 - \text{P\_DIR}_i)$). Now, the attacker loses capital in the second and fourth scenarios, i.e. with probability $\text{P\_ASSET}_{real} * (1 - \text{P\_DIR}_{real}) + \sum_i(\text{P\_ASSET}_i * (1 - \text{P\_DIR}_i))$ and gains capital in only the first scenario with probability $\text{P\_ASSET}_{real} * \text{P\_DIR}_{real}$. The auxiliary assets are chosen such that $\sum_i(\text{P\_ASSET}_i * \text{P\_DIR}_i) = 1 - 2\text{P\_ASSET}_{real} * \text{P\_DIR}_{real}$. Earlier $\text{P\_DIR}_i > 0.5$ was chosen, and thus the restriction needs to be set when choosing the asset pairs (because the attacker would always sandwich the more probable direction after choosing the asset). □

To complete the running example, if user wants to trade USDC to buy ETH, where there is a 0.6 probability (P_DIR_{real}) to buy ETH, and amongst all asset pairs traded in AMMs, USDC-ETH pair occurs 20% of the times, then it would choose another asset pair such that $0.2 * 0.6 = 0.2 * 0.4 + \text{P\_ASSET}_i * \text{P\_DIR}_i$. Thus, an asset that could be used by the user is the WBTC-Tether if it occurs 8.8% of time, and with probability 0.55 the direction is biased towards WBTC. (It could have been 10%, 0.6 probability, and so on as long as it satisfies the formula in the claim). After choosing the asset, the user creates the auxiliary transaction as described. According to our claim, if an attacker sandwiches the new transaction, then it would have the same probability for 'success' case (guessing USDC-ETH, and that the transaction is in the direction of ETH) and 'failure' case (guessing the wrong direction of trade for the guessed pair).

Using the claim with Theorem 2, we get the following theorem:

Theorem 3. *Given there exists a set of asset pairs with probability $P\_{\text{DIR}_i} > 0.5$ of it being traded in a given direction, such that $\sum_i (P\_{\text{ASSET}_i} * P\_{\text{DIR}_i}) = 1 - 2P\_{\text{ASSET}_{real}} * P\_{\text{DIR}_{real}}$, where P_{ASSET_i} represents the probability of asset i to be the real asset amongst the set of assets chosen for a pool of users, given that the user owns an asset not involved in the trade as collateral, $\Delta r_Y^B - \Delta r_Y^- > p^+$ and $\Delta a_X^+ + \Delta a_X^- - 2\Delta a_X < 0$, it is dominant for all parties in AnimaguSwap to follow the specification in Section 5.1.3.*

6 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 [20] and Time Lock Encryption [32] 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 Appendix E can be considered as an implementation of witness encryption assuming trusted hardware.

The notion of traceable secret sharing introduced by Goyal et al.[21] 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 by producing only the generated transactions as output.

On sending deniable messages. Recent studies [39] demonstrate that deniability may be compromised when keys are encumbered in a Trusted Execution Environment (TEE) such as Intel SGX or if a committee manages the flipper's keys through a distributed key system. Consequently, users must verify that they are interacting with a single, unrestricted user as the flipper. This verification can be achieved by employing a Complete Knowledge Proof [24], which substantiates that a single user possesses unrestricted access to the information provided, thereby reinstating deniability. To use CK in practice, all flippers would require a CK certificate either obtained through a TEE (since the input to a TEE is public to the party that inputs it) or an ASIC-based proof (which can be generated in a reasonable time only if the key is known to a party generating the proof) verified on-chain.

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 real-world entity or not. This is reasonable, especially in a permissionless setting. However, in practice, if we can perform an analysis of the flow of transactions across different keys and their uses, and derive intelligence based on these transactions (e.g., [14]), we can identify the existence of such attacks with the analysis acting as a "proof".

On collusion between the user and the flipper. Even if the user and flipper collude, in AnimaguSwap, it is not possible for a user's transaction to violate the soundness condition. During a collusion there are two cases that may arise: the user does not receive a commitment itself, or the flipper violates the commitment it shared. The protocol specification requires the user to collect a commitment before posting a transaction. If the commitment is not given to the user, and the user does not post a commitment to what it receives from the flipper, the flipper can refuse to share profits with the user and instead collude with the committee for a potentially larger share of the profit. Thus, if the user and flipper collude, and the user has the flipper's commitment, the user can not only get better execution and share profits with the flipper but also slash the flipper.

On user acting as flipper. The user cannot be asked to withhold the information in AnimaguSwap. This is because the user cannot be trusted to release the information. As with user withholding information, if the user does not release a flip bit, it cannot be slashed and thus threatens the protocol's liveness. Flipper can, however, be a user (which has a designated stake from being a Flipper), since the transactions being sent by the Flipper would only further discourage sandwich attack since even when the committee is randomly predicting, the flipper can ensure that the prediction they choose is incorrect.

REFERENCES

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

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

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

[4] atom_crypto. 2022. The MEV Game of the Crypto Economy: Osmosis' Threshold Encryption vs. SGX of Flashbot? https://mirror.xyz/infinet.eth/SFjR1H1-RMnKoIoPjqkxpauVPrLYGqLHQP1dY9FHvx4. (2022). Retrieved Oct 6, 2022 from.

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

[6] [n. d.] Balancer. https://docs.balancer.fi/reference/math/stable-math.html. ().

[7] Richard Barnes, Karthikeyan Bhargavan, Benjamin Lipp, and Christopher A. Wood. 2022. Rfc 9180: hybrid public key encryption. (2022). https://datatracker.ietf.org/doc/rfc9180/.

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

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

[10] Bitcoin Wiki. 2021. Payment channels. [Online; accessed 11-November-2021]. (2021).

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

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

[13] Cducrest. 2022. Shutterized beacon chain. https://ethresear.ch/t/shutterized-beacon-chain/12249. (Mar. 2022).

[14] Chainalysis. [n. d.] Chainalysis. https://www.chainalysis.com/. ().

[15] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. (June 2019), 185–200. DOI: 10.1109/EuroSP.2019.00023.

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

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

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

[19] FlashBots. 2020. Flashbots resource document. https://docs.flashbots.net/. (2020).

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

[21] Vipul Goyal, Yifan Song, and Akshayaram Srinivasan. 2021. 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*. Springer, 718–747.

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

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

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

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

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

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

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

[29] 2022. Mev-Boost GitHub. https://github.com/flashbots/mev-boost. (2022). Retrieved Oct 31, 2022 from.

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

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

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

[33] Ronald L. Rivest, Adi Shamir, and Yael Tauman. 2001. How to Leak a Secret. In *Advances in Cryptology — ASIACRYPT 2001*. Colin Boyd, editor. Springer Berlin Heidelberg, Berlin, Heidelberg, 552–565. ISBN: 978-3-540-45682-7.

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

[35] Sikka inc. 2022. Sikka Projects. https://sikka.tech/projects/. (2022). Retrieved Oct 6, 2022 from.

[36] [n. d.] Understanding curve v1 curve finance. https://resources.curve.fi/base-features/understanding-curve. ().

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

[38] Vbuterin. 2022. Secret non-single leader election. (Jan. 2022). https://ethresear.ch/t/secret-non-single-leader-election/11789.

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

[40] Matheus Venturyne Xavier Ferreira and David C. Parkes. 2023. Credible Decentralized Exchange Design via Verifiable Sequencing Rules. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing* (STOC 2023). Association for Computing Machinery, New York, NY, USA, (June 2023), 723–736. ISBN: 978-1-4503-9913-5. DOI: 10.1145/3564246.3585233.

[41] Sen Yang, Fan Zhang, Ken Huang, Xi Chen, Youwei Yang, and Feng Zhu. 2022. SoK: MEV countermeasures: theory and practice. *arXiv preprint arXiv:2212.05111*.

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

A CAPTURING EXISTING SYSTEMS IN THE FRAMEWORK

In this subsection, we show that our framework can capture ordering policy enforcement protocols based on DKG [13, 8], secret-sharing [28], as well as fair ordering protocols [26, 12, 27]. We use (SS.share, SS.rec) to denote the threshold secret sharing and reconstruction algorithms [34], and (Enc, Dec) to denote encryption and decryption.

We use (SS.share, SS.rec) to denote the threshold secret sharing and reconstruction algorithms [34], and (Enc, 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 [30] (without Proposer Builder Separation [11]). Figure 4 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, ShouldReveal(_) is always false since transactions have been revealed in the inclusion phase.

Threshold encryption based content-oblivious ordering. Content-oblivious ordering can be enforced by threshold encrypting user transactions, as in, e.g., Ferveo [8] and Shutterized Beacon Chain [13]. Figure 5 presents their specification in our framework.

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

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

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

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

Figure 4: The specification of the transaction ordering process in PoS Ethereum using our framework.

At a high level, in such systems, stakers run a Distributed Key Generation (DKG) protocol to generate a key pair (sk, 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. $\mathsf{ShouldReveal}(B)$ 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:
- $\mathsf{submit}(\mathsf{tx}, \_) \rightarrow (\mathsf{H}(\bar{\mathsf{tx}}); (\perp, \bar{\mathsf{tx}}, \cdots, \bar{\mathsf{tx}})$ where $\bar{\mathsf{tx}} = \mathsf{Enc}(\mathsf{pk}, \mathsf{tx})$, i.e., stakers do not provide input and receive encrypted tx.
- $\mathsf{process}(\mathsf{txid}, \bar{\mathsf{tx}}, \mathsf{state}_i) \rightarrow \mathsf{state}_i \cup \{(\mathsf{txid}, \bar{\mathsf{tx}})\}$.

Transaction inclusion: Whenever $\mathsf{ShouldRelease}(s_i)$ is true, order does the following
- staker s_i chooses $\mathsf{B} = \{(\mathsf{txid}_i, \bar{\mathsf{tx}}_i)\}_{i=1}^{\ell}$ from state_i,
- s_i adds $(\bar{\mathsf{tx}}_1, \ldots, \bar{\mathsf{tx}}_\ell)$ to the blockchain,
- each staker s_i updates $\mathsf{state}_i = \mathsf{state}_i \setminus \mathsf{B}$.

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

Secret-sharing based content-oblivious ordering, e.g., Fino [28]. 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 6 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, $\mathsf{process}(\mathsf{txid}, (\mathsf{ct}, \mathsf{ck}_i), \mathsf{state}_i) \rightarrow \mathsf{state}_i \cup \{(\mathsf{ct}, \mathsf{ck}_i, \mathsf{false})\}$. 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 $\mathsf{B} = (\mathsf{ct}_1, \ldots, \mathsf{ct}_\ell)$ 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 6: The specification of secret-sharing-based content-oblivious ordering protocols using our framework.

Receive order fairness schemes, e.g., Themis, Aequitas, Quick Order Fairness [27, 26, 12]. 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 submit() protocol, the user simply sends the transaction to all stakers in secure channels. The stakers then apply the 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 [26] and Aequitas [27]) only need relative ordering, which is simplified by timestamps.) Finally, ShouldReveal(_) is always false since transactions have been revealed in the inclusion step. Figure 7 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:

- 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.

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

Transaction submission:
- $\mathsf{submit}(\mathsf{tx}, (\mathsf{pk}_1, \ldots, \mathsf{pk}_N)) \rightarrow (\mathsf{H}(\mathsf{tx}); (\bar{\mathsf{tx}}_i, \ldots, \bar{\mathsf{tx}}_N))$ 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 $\mathsf{B} = (\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell)$.
- 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 7: The mapping of fair ordering protocols to our framework.

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. Figure 8 shows the specification more formally.

C PROOF FOR LEMMAS IN SECTION 4.2

LEMMA 6. *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 state of 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 $\mathcal{P}$. □

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 7. *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 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 6.

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.

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:

```
submit((tx, k), (pk_1, ..., pk_N))
    ∀i ∈ [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 $\pi_i$ be the membership proof of $\mathsf{txss}_i$.
    Send $OP = (r_{TS}, \pi_i, \mathsf{Enc}(\mathsf{pk}_i, \mathsf{txss}_i))$ to each staker $s_i$.

process(txid, OP, state_i)
    Parse $OP$ as $(r, \pi, c)$. Add $(r, \pi, c, \mathsf{false})$ to $\mathsf{state}_i$. 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 $\mathrm{B} = (r_1, \ldots, r_\ell)$ 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 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
    (sk, pk) ←$ KGen(1^λ)
    Seal sk to disk
    return pk_i with hardware attestation.

func Reveal(π_publication, r_TS, π_membership, c):
    txss_i = Dec(sk_i, c)
    Verify π_publication for r_TS, and π_membership of tx_i w.r.t. r_TS
    return txss_i
```

Figure 8: The specification of a secret-sharing-based content-oblivious ordering protocol using TEEs.

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 $\mathsf{tx}'_i \notin$ 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 $((\mathsf{md}_1, \mathsf{data}_1), \ldots, (\mathsf{md}_\ell, \mathsf{data}_\ell))$, where $\mathcal{A}'$ were responsible for receiving, inclusion and revealing the transaction (only a subset of them may be required to attack).

□

The proofs for both the above lemma rely on the fact that no party can generate a proof differentiating two worlds, one where $\mathcal{A}$ deviated from the OPE scheme and one where they followed the OPE scheme.

Lemma 8. *If there exists an* 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 6 and 7 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′. □

With the above lemma, we know that any non-attacking staker entity does not have enough incentive to generate any proof of deviation from the protocol. With both Lemmas 6 and 7, 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).

Proof of Theorem 1. From Lemma 8, 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. □

D EXAMPLE ATTACK ON DKG-BASED THRESHOLD ENCRYPTION

Algorithm 2

E 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

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 $\bot$    ▷ $\mathsf{state}^a$ 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 $\bot$    ▷ $\mathsf{inp}^a$ 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 $\bot$    ▷ $\mathsf{spri}^a$ is a list of secret shares $\mathsf{sk}^j$ for every $s_j$ in $\mathcal{A}$
6:  procedure ATTACK$^{\mathsf{K}}$($\mathsf{state}^a, \mathsf{spri}^a$)    ▷ Executed when 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 := $\textsc{dec}$ - 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:     VerifySigs(B)
13:     att_B $\leftarrow$ extract(B)    ▷ Get MEV-extracting transactions
14:     $\mathsf{state}' \leftarrow \bot$
15:     for att_txn $\in$ att_B do
16:         $(\mathsf{txid}; (\bot, \mathsf{out}_1, \ldots, \mathsf{out}_N)) \leftarrow \textsc{enc}(\mathsf{att\_txn}, \mathsf{inp}^a)$    ▷ submit := $\textsc{enc}$ - 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:     return($\mathsf{tSeq}', \mathsf{state}'$)    ▷ Publish the block containing the MEV-extracting transaction
```

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 [28, 8], 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 [3], AMD SEV [2], and Nvidia H100 [22] 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 [3], 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 8 specifies the protocol following the framework defined in Figure 1. Now we describe the protocol. To submit a transaction tx, a user first 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 [15] 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.

F SANDWICH ANALYSIS

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

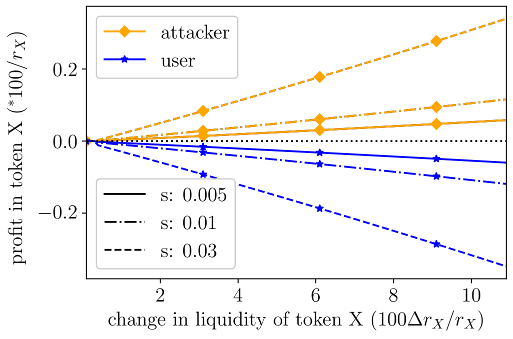


Figure 9: Gains of the attackers and users in a sandwich attack on a vanilla AMM.

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

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 [23, 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

$$\Delta a_X = \frac{\frac{\sqrt{\eta}}{1-s} - \Delta r_X(1-f)^3 - (2-f)(1-f)r_X}{2(1-f)^2} \tag{14}$$

For ease of analysis, we define another placeholder variable,

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

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

Based on the optimal sandwich attack input from [23], we show the results graphically in Fig. 9.

G REPEATED GAME ANALYSIS

AnimaguSwap provides a game where interactions between players occur repeatedly and randomly (with the likelihood of these interactions being independent of past interactions), converting it into a single instance game (where players interact just once) requires making the identities of the players anonymous. This means that in each interaction, players do not know who they are interacting with, making each interaction effectively independent and similar to a one-time game. The major issue is that both parties have a rational interest in de-anonymization, i.e. revealing their identities to collude.

We first design an anonymous flipper selection. The task is similar to a secret non-single leader election (SnSLE) [38]. Stakers generate a hash of a Verifiable Random Function (VRF) (like RANDAO reveal in Ethereum PoS) of their block. Each staker with a value less than $\frac{2^{256}*5}{N}$ as mentioned in the blog [38], would create an alternate key derived from their staker key. Using this identity, in some previous blocks, with some deadline to claim flipper, the staker with minimum value posted for the block would be chosen as the staker. During its interactions to collect transactions, the flipper would use the newly generated identity, however during the reveal phase, the flipper would reveal its staked identity. In this protocol, any honest staker chosen as Flipper is anonymous due to the construction of the protocol.

However, as mentioned earlier, the flipper is incentivized to reveal this information to the committee and collude to extract profits. To de-anonymize the game, the flipper would need to release some information that links it to a previous game (e.g., its real-world identity). We first argue that the previous game linked needs to be the block flipper was last appointed in. This is because if it is linked to a flipper's block before the previous, then the flipper would cooperate in one game and then cheat in the subsequent games, and always show a link to the first game it cooperated in. This allows the flipper to prove that it cooperated in the first game, and then use the rational action of betraying to incentivize itself. Due to the anonymity of the leader election mechanism, no other party than the flipper can prove that the flipper was assigned a particular block or not. Thus, in any link that the flipper is giving, it can choose to skip over the previous block (where it likely betrayed the committee).

Even if some flipper can prove its link to a designated previous block and do so in a way that the proof is non-transferable, we use the following slashing mechanism: the slasher needs to state (and not prove) a link between a previously played game to the current game. If both games have the same flipper, the flipper gets slashed, otherwise, the slasher would lose utility for false accusation which would be required to challenge the flipper.

Using an anonymous flipper with the presented slashing mechanism would be enough to ensure that the flipper would not be incentivized to de-anonymize and collude with the committee to form a multi-shot game.

H EXAMPLE SWAP SCENARIO - DETAILS OF EXECUTION

Table 2

I INTEGRATED ANIMAGUSWAP FOR NON-UNIFORM {Buy, Sell} AND REPEATED GAMES

Figure 12

J 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

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

Table 2: An example swap where the Attacker sees: SELL 8592 SUSHI (S) for WETH (W), when initial reserve of SUSHI is 164467.64, WETH is 73.83, with a 1% slippage. Detailed working of the example is shown in Table 3. (Values of liquidity available pulled on June 3, 2023)

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 3: 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**).**

stakers. What if we have a 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 the impossibility of obtaining OPE.

The intuitive reason that a protocol similar to the one presented in Section 5.1 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 [16]. 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

```
On-Chain Contract :
 Interface IAnimaguSwap:
 function deposit(_amount) payable:
     Registers the caller of the function as a staker (and locks stakes)
 function commit(_hashMerkleRoot):
     User sends commitment to the chain
 function revealFlipper(b):
     Flipper reveals the flip bit b
 function revealStaker(token1,token2,amountIn, _hashWV):
     Reveal details about the sell transaction that was secret-shared to it
     token1, token2, b (from revealFlipper) and amountIn determine the
        direction of the transaction.
     _hashWV is used in case the flipper decides to release a wrong flip
        bit.
     Hash of all inputs to this function are verified against _hashMerkle-
        Root.
 function complain(signed message, v, w):
     If the function is called, the user suspects Flipper for cheating.
     _hashWV is verified against hash(v||w).
     signature of Flipper is checked on a message committing to reveal
        the bit $B \neq b$
     The message must contain $v$.
```

Figure 10: AnimaguSwap On-Chain Contract.

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 $\text{mpc\_inp}_i = (\text{state}_i, \text{spri}_i)$ (which will be used in Algorithm 1) is correct. This lets other stakers trust 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 7 would no longer be true, since now the presence of 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 10, 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 13, 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 5, 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 18, 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 24, Algorithm 3), which is eventually used to check whether or not the MPC transaction sequence has been confirmed on-chain or not (checkConfirmed). Whenever ShouldReveal(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 27, Algorithm 3). Finally, when all transactions have been marked, s_i calls get_preimage (Line 32, 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 9, 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 12, Algorithm 3) to burn all the amount stored in the contract, and take a small transaction fee (ϵ) from the burnt amount.

Lemma 9. *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

User intends tx_0 = BUY(token1,token2,amountOut, amountInExpected, slippage)= swapTokensForExactTokens(amountOut,AmountInMax,path...)
amountInMax is computed by amountInExpected and slippage. path = [token1,token2]
$\cong$ tx_1 = SELL (token2,token1,amountIn,amountOutExpected,s)=swapExactTokensForTokens

Initialization

- deploy the smart contract of AnimaguSwap. All stakers and flipper deposit assets

Generate Transaction:

- Tx = BUY *or* SELL
- b = RandomBit()
- $\sim$ tx = (tx == BUY)? SELL : BUY
- tx_b = (b == 0)? tx :$\sim$ tx

Transaction submission:

- v = RandomBit()
- User to $\mathcal{F}$: $\mathrm{out}_{\mathcal{F}}, \perp \longleftarrow \mathrm{Enc}_{\mathrm{pk}_{\mathcal{F}}}(\mathrm{b}, \mathrm{v})$: user encrypts (b, v) via the public key of flipper and sends it to flipper
- $\mathcal{F}$ to user: On receive $\mathrm{out}_{\mathcal{F}}$; $\mathrm{out}_{\mathrm{u}} \longleftarrow (\mathrm{b}, \mathrm{v})_{\sigma(\mathrm{pk}_{\mathcal{F}})}$: flipper decrypts out_F, signs it and sends it to user as a commitment = Out_U
- User: Upon receive $\mathrm{out}_{\mathrm{u}}$, execute verify(message, σ, wallet_address)
- User to $(s_1 \ldots, s_N)$: (txid; $(\perp, ss_1, \ldots, ss_N)$
  . $\longleftarrow$ SS.share(tx_b, $(\mathrm{pk}_1, \ldots, \mathrm{pk}_N)$) The User uses secret sharing to txb
  ▷ Where ss_i represents the secret share for s_i
- calculate corresponding $MerkleProof_i$ for every ss_i , sign it and send it to every staker
- Stakers use the signed ss_i and $proof_i$ to verify offchain to prevent if user cheats

Transaction inclusion :

- w = RandomBit()
- user commit ($hash(merkleroot)$, $hash(w, v)$: The user calculates the merkle root and hashes the merkle root and the hash of (w, v), submiting them on the chain.

Transaction revealing :

- Every Staker call revealStaker(ss_i, $proof_i$): every staker call revealStaker to use the verify function of merkleProof.sol to verify if staker cheats
- Flipper call revealFlipper(b): reveal b to the blockchain
- User complain (Out_U = $signed(b|v)$,v,w): first verify the hash of v and w, and then compare the Out_U.
- forward both tx and tx_f to AMM for transaction

Figure 11: AnimaguSwap specification.

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 ShouldReveal(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 9, Algorithm 3). □

LEMMA 10. *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 9 still holds. □

LEMMA 11. *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* 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 28, 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 12. *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 = $\{\bar{tx}_1, \ldots, \bar{tx}_\ell\} \in \mathcal{P}(md_1, \ldots, md_\ell)$ for input stream $((md_1, data_1), \ldots, (md_\ell, data_\ell))$. Moreover, let us assume that there exists a function* extract() *known to all stakers such that tSeq′ $\models$* extract$(\mathsf{tx}_1, \ldots, \mathsf{tx}_\ell)$ *and tSeq′ $\in \mathcal{P}(md'_1, \ldots, md'_{\ell'})$ for input stream $((md'_1, data'_1), \ldots, (md'_{\ell'}, data'_{\ell'}))$.*

User intends tx_0 = BUY(token1,token2,amountOut, amountInExpected, slippage)= swapTokensForExactTokens(amountOut,AmountInMax,path...)
amountInMax is computed by amountInExpected and slippage. path = [token1,token2]
$\cong \text{tx}_1$ = SELL (token2,token1,amountIn,amountOutExpected,s)=swapExactTokensForTokens

Given real = {token1,token2}, $\text{P\_DIR}_i, \text{P\_ASSET}_i \forall i \in$ assets in the world

Initialization

- deploy the smart contract of AnimaguSwap. All stakers deposit assets
- For the election of the flipper in block block_num, all stakers generate a Verifiable Random Function according to information in block_num- 2t, where t is a parameter chosen for censorship resistance to ensure fair election of flipper.
- Any staker which gets a value of less than $\frac{2^{256}*5}{N}$ is a potential flipper. It generates a new identity ($\mathcal{F}$') derived from the staker's secret key.
- Using the new identity($\mathcal{F}$'), the potential flipper posts the value of VRF online in the contract.
- Any staker can challenge the proof sent by $\mathcal{F}$'
- On block_num- t, the deadline for posting the VRF expires, and the staker with the lowest value is chosen as the flipper.

Generate Transaction:

- Tx(params) = BUY(params1) or SELL(params2)
- b = RandomBit()
- $\sim$ tx = $(tx ==$ BUY$)?$ SELL : BUY
- $\text{tx}_b = (b == 0)?\, tx(\text{params}) :\sim \text{tx}(\text{params})$
- pool = $\{u, u_1, u_2, ...\}$, s.t. no user is more likely than the other.
- assets = $\{1, 2, 3, ..., i, ..., \text{N\_ASSET}\}$ are the assets chosen such that $\sum_i(\text{P\_ASSET}_i * \text{P\_DIR}_i) = 1 - 2\text{P\_ASSET}_{real} * \text{P\_DIR}_{real}$.
- b_i = RandomBit(P_DIR_i)
- $\text{tx}_{aux-i}(params_i) = (randomcoin_i == 1)?$ SELL$(params0_i)$: BUY$(params1_i)$
- $\sim \text{tx}_{aux-i} = (tx_{aux-i} ==$ BUY$)?$ SELL : BUY
- $\text{tx}_{aux-i-b} = (b == 0)?\, tx_{aux-i}(\text{params}) :\sim \text{tx}_{aux-i}(\text{params})$
- $\text{tx}_b = \text{tx}_b, \text{tx}_{aux-i-b}$

Transaction submission:

- v = RandomBit()^{256}
- User to $\mathcal{F}'$: $\text{out}_{\mathcal{F}'}, \perp \leftarrow \text{Enc}_{\text{pk}_{\mathcal{F}'}}(User, \text{b}, real, \text{v})$: user encrypts $(User, \text{b}, real, \text{v})$ via the created identity key of flipper and sends it to flipper
- $\mathcal{F}$ to user: On receive $\text{out}_{\mathcal{F}'}$; $\text{out}_\text{u} \leftarrow (User, \text{b}, real, \text{v})_{\sigma(\text{pk}_{\mathcal{F}'})}$: flipper decrypts out_F, signs it and sends it to user as a commitment = Out_U
- User: Upon receive out_u, execute verify(message, σ, wallet_address)
- User to $(s_1 \ldots, s_N)$: (txid; $(\perp, ss_1, \ldots, ss_N)$
  . $\leftarrow$ SS.share(tx_b, $(\text{pk}_1, \ldots, \text{pk}_N)$) The User uses secret sharing to tx_b
  ▹ Where ss_i represents the secret share for s_i
- Calculate corresponding $MerkleProof_i$ for every ss_i , sign it and send it to every staker
- Stakers use the signed ss_i and $proof_i$ to verify off-chain to prevent if user cheats

Transaction inclusion:

- w = RandomBit()^{256}
- user commit $(hash(merkleroot), hash(w, v)$: The user calculates the merkle root and hashes the merkle root and the hash of (w, v), submiting them on the chain.

Transaction revealing:

- Every Staker call revealStaker(ss_i, $proof_i$): every staker call revealStaker to use the verify function of merkleProof.sol to verify if staker cheats
- Flipper call revealFlipper(b): reveal b to the blockchain
- User complain ($Out_U = signed(b|v)$,v,w): first verify the hash of v and w, and then compare the Out_U.
- forward both tx and tx_f to AMM for transaction

Figure 12: Integrated AnimaguSwap specification.

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 by running the above SGX code and the contract (Algorithm 3) in addition to Algorithm 1, when the input stream was* $((md_1, data_1), \ldots, (md_\ell, data_\ell))$.

PROOF. The only difference between the above stated lemma and Lemma 7, is that there exists an online contract, publicly visible to everyone. Since the contract has been designed as a Hashed Time Lock Contract (HTLC) [10], 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 13. *If there exists an* extract *function known to stakers such that tSeq'* $\models$ extract$(\text{tx}_1, \ldots \text{tx}_\ell)$*, 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 at least 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 7, 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′. □

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

```
1: ($\mathcal{K}$, ENC, DEC): defines an asymmetric encryption scheme
2: $\mathcal{H}$: represent a cryptographic hash function
3: State
4: secret: A secret revealed when correct mpc_inp$_i$ is verified
5: sk: stores a Secret Key generated inside TEE
6: inp: stores the committed mpc_inp$_i$ value for $s_i$
7: curr_tSeq: stores the last on-chain transaction sequence (block)
8: mpc_tSeq: stores the transaction sequence published on-chain (for which MPC was supposed to happen)
9: block_hash: block hash for the predecessor of MPC block.
10: function KEYGEN
11:     sk, pk ⟵ $\mathcal{K}$()
12:     return pk
13: function GENERATE_HASH(mpc_inp$_i$, _block_hash)
14:     secret = Random()
15:     inp = mpc_inp$_i$
16:     block_hash = _block_hash
17:     return $\sigma$ = sign$_{sk}$($\mathcal{H}$(secret), inp, block_num)
18: function UPDATE_MPC_BLOCK(tSeq)
19:     Assert tSeq.predecessor = block_hash
20:     mpc_tSeq = tSeq
21:     for tx ∈ mpc_tSeq do
22:         if tx.txid ∉ mpc_inp$_i$.state.txid then
23:             mark(tx, mpc_tSeq)        ▷ Mark adds a mark on tx in the variable mpc_tSeq
24: function UPDATE_BLOCK(tSeq)
25:     Verify tSeq is successor of curr_tSeq
26:     curr_tSeq = blk
27: function FEED_REVEALED(tx, B)
28:     Assert CheckInfo(inp, tx)
29:     Assert CheckMembership(tx, B)
30:     if tx.txid ∈ mpc_tSeq then
31:         mark(tx, mpc_tSeq)
32: function GET_PREIMAGE
33:     Assert checkConfirmed(mpc_tSeq)
34:     for tx ∈ mpc_tSeq do
35:         if existsMark(tx, mpc_tSeq) then
36:             return null
37:     return secret
```

Algorithm 4 A contract to add rational binding in the attacking protocol -Contract Side

```
1: State
2: amount_stored ⟵ 0: amount stored in the contract
3: hash ⟵ null: hash of a secret the staker who is committing receives after running generate_hash
4: committer ⟵ null: identity of the staker that commits to the information
5: function COMMIT(amount, hash)
6:     amount_stored ⟵ amount_stored + amount
7:     hash = hash
8:     committer = sender
9: function REFUND(secret)
10:     if $\mathcal{H}$(secret) = hash then
11:         send(amount_stored, committer)
12: function CONFISCATE(timeout)
13:     if current.time > timeout then
14:         burn(amount_stored)
```