Forking the RANDAO: Manipulating Ethereum's Distributed Randomness Beacon

Ábel Nagy*
Eötvös Loránd University
Budapest, Hungary

János Tapolcai†
University of Technology and Economics
Budapest, Hungary

István András Seres‡
Eötvös Loránd University
Budapest, Hungary

Bence Ladóczki§
University of Technology and Economics
HUN-REN Information Systems Research Group
Budapest, Hungary

ABSTRACT

Proof-of-stake consensus protocols often rely on distributed randomness beacons (DRBs) to generate randomness for leader selection. This work analyses the manipulability of Ethereum's DRB implementation, *RANDAO*, in its current consensus mechanism. Even with its efficiency, RANDAO remains vulnerable to manipulation through the deliberate omission of blocks from the canonical chain. Previous research has shown that economically rational players can withhold blocks – known as a block withholding attack or selfish mixing – when the manipulated RANDAO outcome yields greater financial rewards.

We introduce and evaluate a new manipulation strategy, the RANDAO forking attack. Unlike block withholding, whereby validators opt to hide a block, this strategy relies on selectively forking out an honest proposer's block to maximize transaction fee revenues and block rewards. In this paper, we draw attention to the fact that the forking attack is significantly more harmful than selfish mixing for two reasons. Firstly, it exacerbates the unfairness among validators. More importantly, it significantly undermines the reliability of the blockchain for the average user by frequently causing already published blocks to be forked out. By doing so, the attacker can fork the chain without losing slots, and we demonstrate that these are later fully compensated for. Our empirical measurements, investigating such manipulations on Ethereum mainnet, revealed no statistically significant traces of these attacks to date.

1 INTRODUCTION

Randomness is indispensable for (permissionless) consensus [15, 20]. Ethereum [31], the second-largest cryptocurrency by market capitalisation and the largest by transaction volume, pseudo-randomly selects block proposers using cryptographic algorithms and protocols. New validators are chosen using a DRB called RANDAO, introduced in [33]. While the RANDAO is an efficient DRB protocol, it is not robust against strategic manipulations [1, 29].

In Ethereum Proof-of-stake (PoS), blocks are published at fixed time intervals (*i.e.*, at every 12 seconds), referred to as slots, with 32 slots comprising one epoch. The RANDAO mechanism selects validators who publish the blocks in the subsequent epoch on the epoch boundaries (*i.e.*, at every 384 seconds). In each slot, there is only one uniquely determined validator who should publish a block. Should it fail to do so, the slot is marked as missed in the canonical chain. As the output of the randomness beacon depends on these missed slots, this creates an opportunity for an attack. Validators colluding within staking pools might find it profitable to intentionally miss blocks if this influences the RANDAO outcome such that, in upcoming epochs, more slots are allocated to validators of the given staking pool. The effect of these missed slots is only predictable before the epoch boundaries, at the so-called tail slots.

The first analysis on the manipulability of the current RANDAO protocol was conducted in a blog post by Wahrstätter [29]. He examined the aforementioned attack (selfish mixing) and demonstrated that major staking entities could have manipulated RANDAO dozens of times since the genesis of Ethereum PoS by strategically publishing or omitting tail slots. Alpturer and Weinberg, applying Markov Decision Processes (MDP), determined optimal selfish mixing strategies in [1] to maximise the number of blocks proposed. According to previous results, a staking pool controlling 20% of the total stake can gain an additional 0.7% of slots with selfish mixing [1] and when a strategic player mounts a selfish mixing attack roughly every 340^{th} slot will be missed, causing a 0.3% reduction in transaction throughput. This can potentially increase transaction costs due to network congestion [8]. However, such issues alone do not pose a threat to the operation of the blockchain.

In this paper, we introduce a new class of RANDAO manipulation strategies that pose a much greater threat to the average user. Our main observation is that a strategic validator can fork out honest (tail) blocks from the canonical chain to manipulate the RANDAO outcome. First, this manipulation significantly increases the bias in the DRB, potentially doubling the attacker's gain in extra slots when combined with selfish mixing. What further exacerbates the situation is that it undermines trust in the chain's ecosystem. By forking blocks, transactions in the discarded blocks are removed, and the attacker can hijack the Maximum Extractable Value (MEV) of that block.

Forking is already a serious issue, but it becomes even more severe when linked to strategic RANDAO manipulations. As we will discuss later, in some epochs, the attacker has a direct incentive to fork out other validators' blocks, as this provides short-term profits in addition to more MEV.

We argue that malicious players can perform short-range forks when they have multiple adversarial slots surrounding honest blocks. The attack is initiated as follows: The adversary privately

*nagyabi@gmail.com
†tapolcai@tmit.bme.hu
‡seresistvanandras@gmail.com
§ladoczki.bence@vik.bme.hu

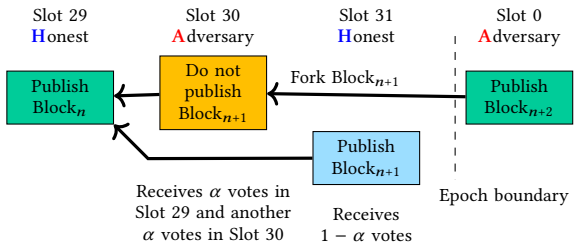


Figure 1: Illustration of an attack where the adversary ($\mathcal{A}$) is selected to propose blocks in Slot 30 of the current epoch and the first slot of the next epoch, with the final slot assigned to an honest validator. This scenario is called AH·A forking attack. Assume $\alpha(\geq 0.2)$ denotes the adversary's voting power, while $1-\alpha$ denotes the honest voting power.
The attack proceeds as follows: First, $\mathcal{A}$ secretly builds and votes for its block in Slot 30. This block is kept hidden from honest validators. Hence, they think that the block in Slot 29 is the head of the chain and vote for it. Second, the honest validator publishes a block in Slot 31 and builds it on top of Slot 29. Meanwhile, the validators controlled by $\mathcal{A}$ vote for the adversary's secret block during Slot 31, pretending to be unaware of the honest block published in Slot 31. Finally, in Slot 0 of the next epoch, $\mathcal{A}$ publishes its two-block fork (illustrated in the first row of the figure), and the honest block in Slot 31 is successfully forked out from the canonical chain [26], thus not contributing to the RANDAO output.

builds a fork and purposefully ignores the next few blocks proposed by the honest players. Finally, the adversary forks out the "surrounded" honest block(s) by building on the private fork. As a side effect, this forking attack leads to slots missing from the canonical chain. In general, blockchain consensus mechanisms might try to thwart forking attacks. However, it is usually not a punishable offense because it sometimes happens naturally due to network delay. This study investigates under what circumstances such an attack can be economically rational. To understand the inner workings of these forking attacks, we must delve into the Ethereum PoS consensus protocol [22]. The current Ethereum PoS protocol has evolved throughout the years: the consensus protocol has been revised multiple times as new practical attacks were discovered, challenging its presumed liveness and safety [26]. The original protocol, LMD-GHOST (Latest Message Driven Greedy Heaviest Observed Sub-Tree) was introduced in [7]. In LMD-GHOST, validators attest to a block in each slot, and the chain with the most accumulated attestations becomes the canonical chain, considering only the latest received messages. To counter block withholding attacks, *proposer boost* has been introduced to artificially increase the weight of a block that has been proposed on time.

We elucidate the details of a forking attack in Figure 1. Note that this attack is possible only when the adversary controls at least 20% of the total stake and when the tail slots are ordered as adversary-honest, with the next epoch starting with an adversarial slot (denoted as AH·A in our notation). This is because, in the LMD-GHOST protocol, the adversary's branch has a weight $2\alpha + p_{\text{boost}}$, (at the time of writing $p_{\text{boost}} = 0.4$) while the honest branch has a weight $1-\alpha$. If $\alpha \geq 0.2$, the $\mathcal{A}$'s branch has more attestations, forcing honest validators to choose that adversarial branch and attest to the block in Slot 0 as the head of the chain. As a result, the honest block proposed in Slot 31 gets forked out.

This example highlights some key differences between forking and selfish mixing. First, the adversary does not necessarily need to control the tail slot; controlling the beginning of the next epoch may be sufficient. This complicates the attack strategy as it depends on the validators' actions across two epochs. In other words, when the attacker manipulates the RANDAO outcome to enable further attacks in future epochs, the necessary preparations for two further attacks are being made, accounting for two simultaneous attacks: one at the beginning of the new epoch and another at the end of that epoch.

This work formalises these RANDAO forking attacks and analyses how much financial gain they lead to by applying these strategic manipulation techniques. Similar to [1], we use a Markov Decision Process (MDP) to compute the optimal strategy (or policy) that an attacker should follow to potentially manipulate the RANDAO outcome. This approach maximises the immediate reward and the opportunities for future attack-related rewards in upcoming epochs as well. In the case of forking attacks, the MDP expands considerably compared to scenarios involving only selfish mixing. This is partly due to the larger degrees of freedom with many possible strategies and the fact that many attacks extend across epoch boundaries. As a result, the states of the MDP extend across two consecutive epochs, creating an extensive directed graph. Evaluating this MDP poses a significant computational challenge and proving its optimality remains unattainable.

Our contributions. We provide the following contributions.

- We create a formal model applying Markov Decision Processes (MDP) to compute the extent (expected number of proposed blocks) to which a strategic player can bias the RANDAO output in its favour *employing both selfish mixing and forking strategies*. Our RANDAO manipulation results are currently the best known; see Table 2.
- We collect and process traces from the beacon chain to investigate whether RANDAO manipulation attacks occur. We search for both one- and multi-epoch RANDAO manipulations. We have not yet found statistically significant evidence for multi-epoch manipulations. However, since the protocol currently lacks cryptographic guarantees to prevent such attacks, they are likely to occur in the future, given that validators act as economically rational agents.
- For the sake of reproducibility, we release all our program codes in an open-source repository. [1]

The rest of this paper is organised as follows. In Section 2, we describe the pertinent parts of Ethereum's proof-of-stake protocol. Section 3 describes the RANDAO manipulation strategies we consider in this work. In Section 4, we formalize and compute the advantage of an adversary applying our RANDAO manipulation techniques. In Section 5, the results of our empirical tests on Ethereum mainnet are presented. We discuss countermeasures in Section 6 and review related work in Section 7. We conclude our paper with open problems and future directions in Section 8.

[1] https://github.com/nagyabi/forking_randao_manipulation.

2 PRELIMINARIES AND SYSTEM MODEL

This section introduces the pertinent parts of the Ethereum proof-of-stake consensus protocol. We focus on randomness generation and leader selection and omit irrelevant protocol details. A comprehensive overview of the full protocol can be found in [22].

2.1 Notations

Arrays are written in lowercase bold, *e.g.*, $\mathbf{v}$. We use a Python-like notation for array elements and slices, *e.g.*, $\mathbf{v}[0]$ or $\mathbf{v}[i:j]$, which is non-inclusive on the right, *i.e.*, $\mathbf{v}[j] \notin \mathbf{v}[i:j]$. We denote counterfactual values—those that are reorged or not part of the canonical chain—with a superscript star ($*$).

For instance, the RANDAO output at epoch e is denoted as R^e. However, due to adversarial bias, it could take on various other counterfactual values, denoted as $R^{e,*}$. Let $X \sim \text{Binom}(N,p)$ denote a binomial distribution with parameters N, p. Typically, we use $(N,p) = (32,\alpha)$. $x \leftarrow S$ denotes sampling x from distribution S. See Table 1 for a summary of the used notations.

2.2 Leader selection in Ethereum

Block proposers in the Ethereum PoS protocol are selected by a distributed randomness beacon called RANDAO. The RANDAO outputs a pseudorandom value $R^e \in \{0,1\}^{256}$ at the end of each epoch e. An epoch consists of 32 slots, we number the slots from 0 to 31, where slot 31 is called the tail slot. In each slot, a single validator, identified by their public key, can propose a block. Each valid proposed block contains a 96-byte value called `randao_reveal`, the BLS signature of the epoch number generated using the proposer's private key. As part of block verification, this value is validated against the proposer's public key. This information cannot be determined until the block is published unless the proposer's private key is compromised. Furthermore, the BLS signature scheme is deterministic [5] to prevent the signer from manipulating it (grinding attacks). The final random output is calculated as the bitwise XOR of the hashes of all `randao_reveal` values from each previous block, all the way back to the genesis block. Formally, the RANDAO beacon output at epoch e is calculated as $R^e := R^{e-1} \oplus \left(\bigoplus_{i=0}^{31} h(r_i^e) \right)$,where $h(r_i^e) \in \{0,1\}^{256}$ is the ith `randao_reveal`'s hash in epoch e. The initial condition is $R^0 := 0^{256}$. Missed blocks are excluded from the calculation, formally a missed block is taken as $h(r_i^e) := 0^{256}$.

The RANDAO output R^e defines the proposers' list[2] in epoch $e+2$. [3] Specifically, the proposer list is shuffled using a pseudo-random permutation [17], where the permutation seed s is further randomised with the epoch number, *i.e.*, $s := h(e||R^e)$. This ensures that even if all proposers selected in the current epoch are offline (*i.e.*, $R^e = R^{e-1}$), the proposer list will differ in the next epoch. Let $\mathbf{v}_R^{e+2}$ denote the proposer list in epoch $e+2$ if the RANDAO output in epoch e is R^e, formally, $\mathbf{v}_R^{e+2} = [F_s(\mathbf{v})[j]]_{j=0}^{31}$. Let us write the *number of blocks* allocated to validator i in a certain epoch e for a

[2]The current protocol is somewhat more complex, but the details are immaterial to our discussions. Specifically, validator effective balances are also taken into account.

[3]Occasionally, we use the terms 'future epoch' or 'next epoch' loosely, even though the current e^{th} epoch's RANDAO output determines the validators in epoch $e+2$ in the protocol.

Table 1: Summary of notations used throughout the paper.

Variable	Description
α	The adversary $\mathcal{A}$'s staking power ($0 \le \alpha < \frac{1}{2}$)
e	Epoch number ($e \in \mathbb{N}$)
r_i^e	A validator's beacon contribution in ith slot of epoch e
R^e	The RANDAO beacon's output in epoch e
$\mathbf{v}_R^e$	The chosen validators in epoch e from R ($\lvert\mathbf{v}_R^e\rvert = 32$)

RANDAO output R^e be as:

$$p(e+2, R^e, i) \ . \tag{1}$$

Observe that $p(e,\cdot,i)$ is a discrete probability distribution in the RANDAO output R^e of epoch e, with a range $0 \le p(e,\cdot,i) \le 32$. Moreover, $p(e,\cdot,i) \sim \text{Binom}(32,\alpha_i)$ assuming cryptographic building blocks are idealised, where α_i is validator i's staking power.

Each slot has a duration of 12 seconds, during which a selected proposer can publish a new block. Due to stringent latency requirements, each block must be published no later than the 4^{th} second of the slot. Otherwise, it is deemed as a "missing" block [16]. If a block is published on time, it receives virtual votes, called proposer boost (p_{boost}). At the time of writing, $p_{\text{boost}} = 0.4$, *i.e.*, a newly published block receives virtual votes that equals 40% of the total stake. Note that the proposal boost only exists for the current slot of the block. Blocks published late do not receive proposer boost.

2.3 System model and manipulation objectives

In our model, we assume two parties, an adversarial entity $\mathcal{A}$ with staking power α (*i.e.*, owns α portion of all validators), and an honest entity $\mathcal{H}$ with staking power $(1-\alpha)$. Based on the RANDAO output R^{e-2} of epoch $e-2$, $\mathbf{v}_R^e$ defines the list of proposers for epoch e. The proposers controlled by the adversary $\mathcal{A}$ (or strategic player) are denoted by $\mathbf{A}$, and the slots assigned to the honest entity $\mathcal{H}$ are denoted as $\mathbf{H}$. Using these notations for the slots, we define a string called the *chain string* (cs), which is a continuously growing string over time. More precisely, at the end of epoch e, we extend cs using a $\{\mathbf{A},\mathbf{H}\}^{32}$ string based on $\mathbf{v}_R^{e+2}$, called the $(e+2)^{\text{th}}$ *epoch string*. When the same character repeats, we typically indicate the number of repetitions in the exponent. From a RANDAO manipulation perspective, the slots immediately before and after the epoch boundary are the most important. The potential attacks will be defined using these string fragments, which we refer to as *attack string*:

Definition 1 (Attack string). *Let $(m,n) \in \mathbb{N}^2$. Let the set of attack strings be denoted by $AS_\alpha(m,n)$, where each attack string as $\in \{\mathbf{A},\mathbf{H}\}^{\le m} + \cdot + \{\mathbf{A},\mathbf{H}\}^{\le n}$, where "$\cdot$" denotes the epoch boundary separating epoch e and $e+1$. We refer to the substring before the epoch boundary as* tail(as) *and the substring after the epoch boundary as* head(as). *$AST_\alpha(m)$ denotes the set of possible tail slots in $AS_\alpha(m,n)$. Additionally, upon reaching the first slot $\mathcal{A}$ can already calculate at least one potential RANDAO outcome $R^{e,*}$. An empty string is denoted by ϵ, and $AS_\alpha(m,n)$ includes an attack string called the honest attack string $\mathbf{H}\cdot$.*

Individual slots of an attack string are referred to by their slot number in the subscript, *i.e.*, the tail slot would be $\mathbf{H}_{31}$ in Figure 1.

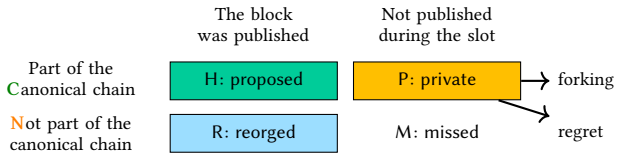


Figure 2: The four block states in a RANDAO manipulation with their colour encodings in the figures. Slot statuses, (non)canonical, are denoted as N, C.

We assume that the stake distributions remain constant throughout the duration of the attack. Furthermore, in each slot, the adversary has exactly α fraction of the voting power. Considering the law of large numbers, this simplification must make sense. The honest entity $\mathcal{H}$ always broadcasts a block when it is selected to do so. Moreover, honest blocks are always built upon the latest public head of the blockchain. In contrast, the adversary $\mathcal{A}$ may miss proposing blocks in the allotted slot. Furthermore, the adversary might strategically fork the blockchain, *i.e.*, propose blocks on blocks that are parents of the current head of the blockchain. We work in a synchronous network model and we assume that the upper bound on message delivery is less than one-third of a slot's duration (4 seconds). Finally, we rely on idealised cryptographic building blocks.

We aim to maximise the number of adversarial blocks. We denote the *number of missed tail blocks* in epoch e (sometimes referred to as sacrificed blocks) corresponding to a RANDAO output R^e as

$$s(e, R^e, \mathcal{A}) \ . \tag{2}$$

An economically rational validator $\mathcal{A}$ applying policy π is incentivised to manipulate the RANDAO output R^e to maximise the expected number of obtained blocks in the future, *i.e.*: [4]

$$\Gamma_\pi := \lim_{N\to\infty} \mathbb{E}\Big[\frac{1}{N}\sum_{e=1}^{N} p(e+2, R^e, \mathcal{A}) - s(e, R^e, \mathcal{A})\Big] \ . \tag{3}$$

This is the most natural RANDAO manipulation objective of a validator, which is a policy to maximise the expected number of proposed blocks without sacrificing too many present gains, *e.g.*, missing to propose too many blocks. We do not use discount factors in our reward function Γ_π, *i.e.*, $\gamma = 1$: slots in the distant future are equally valuable as the ones in the imminent future.

3 RANDAO MANIPULATION STRATEGIES

From the perspective of RANDAO manipulation, a block can have four states, see Figure 2.

- **H: proposed** The validator successfully proposed a valid block accepted by the supermajority of the validators (*i.e.*, 2/3) into the canonical chain. Honest validators always follow this behaviour in our analysis, while adversarial validators may consider additional options, such as withholding blocks.
- **R: reorged** The validator proposed a valid block; however, it ended up on a non-canonical branch of the blockchain. It is no longer considered canonical by the supermajority of the validators' stake, *e.g.*, Slot 31 in Figure 1.

[4] We define the parameter of the reward functions, *i.e.*, our MDP policy in Section 4.1.

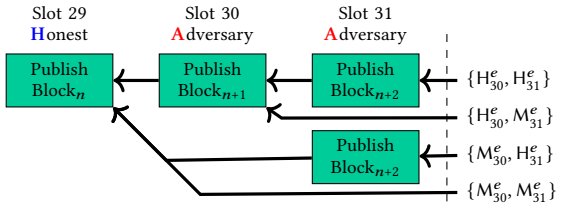


Figure 3: Decision tree for the selfish-mixing with attack string $A^2\cdot$. $\mathcal{A}$ computes R^e for all four scenarios and selects the most lucrative one.

- **M: missed** A validator fails to publish a block in its designated slot. For an honest validator, this is typically caused by connectivity or other operational issues (*e.g.*, large computational overhead due to validating attestations).
- **P: private** The validator builds a block but does not publish it on time during its allotted slot. An adversarial validator then shares the block only with the validators within its staking pool. Later, the private block either becomes part of the canonical chain by *forking* the next block (a strategy known as an ex-ante reorg attack) or, depending on the RANDAO outcome, the attacker might decide not to publish the block at all – an action we refer to as *regret*, see Figure 4.

Block statuses are denoted as $H^e_i, R^e_i, M^e_i, P^e_i$ indicating that the block in the ith slot in epoch e was proposed, reorged, missed, built privately, respectively. As indicated in Figure 2, reorged and missed blocks *do not contribute to the RANDAO output* R^e since they are not part of the canonical chain. In general, RANDAO manipulation strategies can take advantage of this fact. This study examines two strategies—selfish mixing and forking—that enable an adversary to manipulate the RANDAO output.

3.1 Selfish mixing

The adversary can selectively propose or miss blocks to manipulate the RANDAO output [29]. Assume that $\mathcal{A}$ is assigned with t consecutive tail blocks, formally $A^t\cdot$ of epoch e, then $\mathcal{A}$ can choose arbitrarily between 2^t RANDAO outputs by missing or proposing each tail block. Thus it is trivial, that $A^t\cdot \in AS_\alpha(m, n)$ for $0 \le t \le m$, as $\mathcal{A}$ can compute R^e corresponding to C^t. The manipulative power for $t = 2$ is the decision tree shown in Figure 3, *e.g.*, the adversary chooses option $\{H^e_{30}, M^e_{31}\}$ if the calculated R^e *eventually* leads to the highest number of blocks. In this case, sacrificing Slot 31 is worthwhile, as it results in a significantly higher number of blocks in epoch $e + 2$. When evaluating selfish mixing, the adversary has all the necessary information to make decisions. This is not the case for forking, as we will see next, leading to more complex decision trees and evaluations.

3.2 Ex-ante reorging honest blocks

A RANDAO manipulator can selectively reorg out honest blocks from the canonical chain to influence the output of the randomness beacon. The attack strategy can be more complex than just missing tail slots. The simplest forking manipulation is illustrated in Figure 4 when $\mathcal{A}$ is given the attack string $AH\cdot A$, which is the same attack

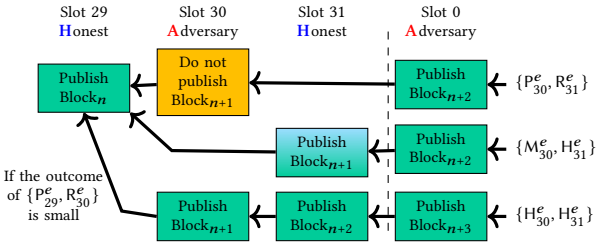


Figure 4: Decision tree for the forking RANDAO manipulation, see Section 3.2 with attack string AH·A. If $\mathcal{A}$ has $\alpha > 0.2$, it can choose from three scenarios, each with different RANDAO outputs. Note, $\mathcal{A}$ needs to make the decision before seeing the honest party's RANDAO contribution r^e_{31}. Consequently, $\mathcal{A}$ may regret its decision.

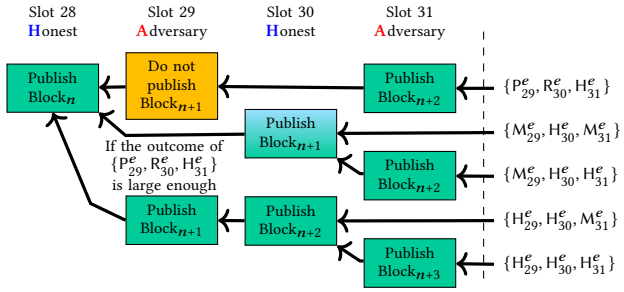


Figure 5: Decision tree for the forking and selfish mixing RANDAO manipulation with attack string AHA· and $\alpha > 0.2$. $\mathcal{A}$ needs to make two decisions: first, before seeing the honest party's RANDAO contribution r^e_{30}, and afterwards.

described in Figure 1. In this scenario, $\mathcal{A}$ can ex-ante fork out the tail slot H if $\alpha \geq 0.2$. $\mathcal{A}$ must decide to fork out H before seeing its `randao_reveal` r^e_{31}. After seeing r^e_{31}, $\mathcal{A}$ can reconsider its forking plans. Specifically, $\mathcal{A}$ can decide in the first slot of the next epoch to build A_0 on top of either A_{30} (*i.e.*, finalise the ex-ante reorg), or on top of H_{31} incurring the sacrifice of A_{30}. Note $\mathcal{A}$ has no incentive to withhold A_0 as it holds no manipulative power.

$\mathcal{A}$ must make a decision in Slot 30, relying solely on the RANDAO output corresponding to $\{P^e_{30}, R^e_{31}\}$. The adversary's strategy involves using stochastic methods to approximate the unknown RANDAO outcome and comparing these approximations with the known values. In other words, if $\{P^e_{30}, R^e_{31}\}$ results in a sufficiently high revenue, $\mathcal{A}$ will privately build the block in its allotted adversarial Slot 30; otherwise, $\mathcal{A}$ will propose it. The exact value of r^e_{31} will only be known after Slot 31, and it is possible that $\mathcal{A}$ may *regret the forking decision* made in Slot 30.

The forking attack becomes truly effective when combined with selfish mixing. This combination occurs recursively, *e.g.*, consider a variation of the previous attack with the epoch boundary shifted by one, resulting in AHA· and $0.2 \geq \alpha$. In this case, as shown in Figure 5, each branch of the decision tree offers an opportunity for an additional selfish mixing attack. This increases the number of possible RANDAO outcomes from three to five as follows:

- $\{H^e_{29}, H^e_{30}, H^e_{31}\}$: Parties publish blocks in their allotted slots.
- $\{H^e_{29}, H^e_{30}, M^e_{31}\}$: $\mathcal{A}$ misses its tail slot to selfish mix.
- $\{P^e_{29}, R^e_{30}, H^e_{31}\}$: $\mathcal{A}$ can reorg the honest block if $\alpha \geq 0.2$. This attack realisation is shown in Figure 1.
- $\{M^e_{29}, H^e_{30}, H^e_{31}\}$: $\mathcal{A}$ foregoes forking after the 30th slot, *i.e.*, $\mathcal{A}$ misses to propose in the 29th slot but publishes in the last slot. Colloquially, we refer to this action as *regret*.
- $\{M^e_{29}, H^e_{30}, M^e_{31}\}$: $\mathcal{A}$ may forego forking after the 30th slot and miss proposing blocks in both of its allotted slots. We refer to this branch (and the previous one) as "regretted forking".

Although there are five possible outcomes, the attacker can choose from at most three, depending on a prior decision. Later, in Appendix A.2, we provide a general description of how to construct decision trees recursively.

Definition 2 (Slot status). *The slot status can be either:*

- C - *Canonical, the slot is proposed or private.*
- N - *Non-canonical, the slot is reorged or missed.*

We refer to the strings $c \in \{C, N\}^{\leq 32}$ *as* realisation strings *corresponding to the* e^{th} *epoch's tail slots' statuses producing a different RANDAO outcome* $R^{e,*}$.

For instance, for A^2·, the four possible realisation strings are $\{C^2, N^2, CN, NC\}$ and for as = AH·A; c = $\{C^3, CNC, NCC\}$.

3.2.1 Ex-ante reorgs in a stronger adversarial model. Ex-ante reorgs are possible for a wider range of attack strings, which we analyse in this paper. However, they require a stronger adversarial model that we refrained from incorporating into our system model. Specifically, the following ex-ante reorg is possible if we assume that network delay is under adversarial control. Evaluating these types of ex-ante reorgs could provide valuable insights for future work. If $\mathcal{A}$ is given tail slots AAH·H, $\mathcal{A}$ can fork out the H tail slot with $\alpha \geq 0.2$ stakes as follows. First, $\mathcal{A}$ secretly builds both its assigned slots and does not publish them. Naturally, $\mathcal{A}$ votes for both blocks; thus, this fork accumulates 2α votes. The honest players do not see this fork; hence, they vote for the first block preceding AA as the head of the blockchain. Moreover, they build H on top of the block proposed in Slot 28. When the block H_{31} is published, it immediately obtains the proposer boost, $p_{\text{boost}} = 0.4$. $\mathcal{A}$ immediately publishes its own secret fork totalling 2α votes. Assuming $\mathcal{A}$ controls the network delay and its fork reaches honest players faster, honest nodes will vote for the adversarial fork as the heaviest subtree despite the virtual proposer boost helping the honest player. Such a robust adversarial model is frequently encountered in the PoS consensus literature [26].

3.2.2 Ex-ante reorg attack strings. As stated in Definition 1, an attack string must meet the condition that by the time $\mathcal{A}$ reaches its first slot it should be able to calculate one possible $R^{e,*}$. This is only possible if the future RANDAO reveals corresponding to the honest blocks do not contribute to $R^{e,*}$ and $\mathcal{A}$ can fork them out. The following theorem describes a condition under which $\mathcal{A}$ can perform ex-ante reorgs in our model. The proof is deferred to Appendix A.1.

Theorem 1 (Condition for forking). *Given* A^{a_1} *slots followed by* HXA, *(where* $X \in \{A, H\}^{h-1}$, $a_1, h > 0$*)* $\mathcal{A}$ *can perform an*

ex-ante reorg with $0 < \alpha < 0.5$ *stakes forking out* $\mathbf{HX}$ *if*

$$a_1 \geq \frac{h(1-2\alpha) - p_{\text{boost}}}{\alpha} \quad . \tag{4}$$

Strings in the form as $\in \mathbf{A}^{a_1}\mathbf{H}^{h_1}\cdot\mathbf{H}^{h_2}\mathbf{A}$ ($h_1 > 0$) are attack strings if Equation (4) is true for $h = h_1 + h_2$. If $\mathcal{A}$ forks out $\mathbf{H}^{h_1+h_2}$, the RANDAO reveals of $\mathbf{H}^{h_1}$ do not contribute to $R^{e,*}$.

Theorem 2 (Recursive attack strings). *Given any* $S \in AS_\alpha$, *for a forking string* $\mathbf{A}^{a_1}\mathbf{H}^{h}\mathbf{A}$ *where Equation* (4) *holds,* $\mathbf{A}^{a_1}\mathbf{H}^{h}\mathbf{A}S$ *is also an attack string,* i.e., $\mathbf{A}^{a_1}\mathbf{H}^{h}\mathbf{A}S \in AS_\alpha$.

Due to space constraints, the proof is deferred to Appendix A.1.

3.3 Ex-post reorging honest blocks

In contrast to ex-ante reorgs, ex-post reorgs allow $\mathcal{A}$ to fork honest blocks out *after they were proposed.* A validator could be motivated to fork out blocks a posteriori if they contain unusually high transaction fees [8]. However, at the time of writing, no reliable technique is known for non-majority entities (*i.e.*, $\alpha < 0.5$) to perform ex-post reorgs [26]. Therefore, we do not consider this forking manipulation in our theoretical model. Consequently, strings starting with $\mathbf{H}$ are not attack strings, as the `randao_reveal` corresponding to $\mathbf{H}$ is computationally hard to derive given that BLS signatures are existentially unforgeable. For convenience, we will make an exception by referring to "$\mathbf{H}\cdot$" as a valid attack string. Nevertheless, ex-post reorgs can occur in practice, see Section 5.2.2 and could be used for RANDAO manipulation. If a validator fails to propose a block in time, it might receive a small fraction ϵ of the attestations. If the subsequent block's proposer sees this and $\epsilon < p_{\text{boost}}$, then $\mathcal{A}$ can fork out the preceding block for any α. At the time of writing, similar mechanics *i.e.*, *honest reorgs* [27] are implemented in the consensus protocol, though following them is optional. Reorging happens when the previous block was proposed late and received few attestations *i.e.*, $\epsilon < 0.2$, except for some edge cases. We do not model honest reorgs in our MDP because a strategic player cannot force honest validators to propose blocks late in our adversarial model.

3.4 Detectability of RANDAO manipulations

Not all RANDAO manipulations are created equal from a detection point of view. In some cases, $\mathcal{A}$ can plausibly deny that they have manipulated the beacon output in their favour.

3.4.1 Detecting the selfish mixing manipulation. If the adversary misses one of its tail blocks, say $\mathbf{A}\cdot$, then one cannot recompute the counterfactual RANDAO output $R^{e,*}$ that also contains the contribution r^e_{31}. This is because r^e_{31} is not publicly available and is hard to compute. Therefore, one cannot decide whether the adversary has missed a block on purpose (*i.e.*, to manipulate the RANDAO for its own gains) or by accident, *e.g.*, network outage. More generally, if the adversary owns k tail slots and misses $n \leq k$ of them, then one can only recompute 2^{k-n} RANDAO counterfactual outputs $R^{*,e}$ out of the 2^k counterfactual outputs that the adversary sees. This greatly limits the detectability of the selfish mixing manipulation strategy. We note, however, that if a validator misses tail blocks enough times, one could apply a binomial statistical test (*e.g.*, Student's t-test, Z-test) to check whether the validator manipulates the RANDAO output for its own gains, see Section 5.1.3.

3.4.2 Detecting manipulations for reorged blocks. Since reorged blocks had been proposed, they contain valid RANDAO contributions. For the sake of concreteness, consider the attack string $\mathbf{AH}\cdot\mathbf{A}$ and suppose the adversary reorgs block $\mathbf{H}_{31}$ in the tail slot with RANDAO contribution r^e_{31} by building its block $\mathbf{A}$ on top of each other. Now, one could recompute how many blocks the adversary would have won with $(R^e \oplus h(r^e_{31}))$ or without (R^e) the honest tail block. Therefore, these RANDAO manipulations are publicly verifiable. See our measurements of these manipulations on Ethereum main net in Section 5.2.1.

4 RANDAO MANIPULATION PROFITABILITY

Moving further, we describe how to devise a stochastic strategy in our RANDAO manipulation model, which considers selfish mixing *and forking strategies.* Finally, we evaluate the efficacy and other properties of this policy.

4.1 RANDAO manipulations strategies

A detailed analysis of RANDAO manipulations, *focusing solely* on the selfish mixing strategy, can be found in [1]. The authors show that RANDAO manipulation is profitable not only because of

- ***immediate rewards:*** it leads to more $\mathbf{A}$ slots in epoch $e+2$, but also because of
- ***future rewards:*** it enhances the attacker's ability to further manipulate RANDAO in epoch $e+2$, potentially yielding additional $\mathbf{A}$ slots in epoch $e+4$, and so on.

Should one disregard future rewards, the attacker's strategy is straightforward: given the opportunity for RANDAO manipulation, simply calculate the possible epoch strings and choose the one that yields the most $\mathbf{A}$ slots in epoch $e+2$, subtracting the missed slots sacrificed in epoch e. To evaluate this strategy one can explicitly calculate the strategy's reward, similar to how selfish mining was evaluated for Bitcoin PoW [12].

However, to account for future rewards an attack strategy has to be calculated. This involves more advanced game-theoretical tools, specifically the Markov Decision Process (MDP). Alpturer and Weinberg [1] point out that for selfish mixing, the optimal strategy is simple to describe due to a massive reduction in state space in the Markov chain. The future reward is an additive value, represented by a utility function for each attack string, which also depends on the stake α. The attacker's best strategy (policy) is to choose the RANDAO outcome that maximises the sum of the immediate rewards and the corresponding attack strings' utility.

The RANDAO manipulation MDP for selfish-mixing is defined as a tuple consisting of *states*, *action space*, *policy*, *transition probabilities*, and *rewards*. The states represent selfish-mixing attack strings, such as $\mathbf{A}^t\cdot$, for $t \in \{0, \ldots, 32\}$. The action space allows the adversary, for each slot $\mathbf{A}$ in the attack string, to choose between proposing (H) or withholding (M) the adversarial slot.

As discussed in [1], the determination of the policy reduces to finding a utility function over the attack strings. The transition probabilities are determined by a game-like process, modeled using off-the-shelf stochastic methods whereby the attacker selects the

RANDAO outcome (the observation) that maximises the sum of the immediate rewards and the utility of the corresponding attack string. Specifically, the RANDAO outcome is sampled by assuming that each slot in epoch $e+2$ is drawn independently, proportional to the stake α. This transition's reward is given by its immediate rewards. Finally, a technique called policy iteration is applied to find a strategy that can be shown to be optimal as it satisfies the Bellman equations.

Next, we overview how the MDP of selfish mixing can be generalised to account for *forking strategies*. The situation is significantly more complex in the case of forking for the following two reasons:

(1) The attack strings may span the epoch boundary *cf.* Figure 4.
(2) Some attacks require decisions before seeing all the possible RANDAO outputs *cf.* Figures 4 and 5.

To tackle 1) we define the *extended* attack string, *i.e.*, a string that also considers the beginning of the next epoch.

Definition 3 (Extended attack string). *For $(m,n) \in \mathbb{N}^2$, the set of extended attack strings $EAS_\alpha(m,n)$ consists of all possible strings*

$$\{\mathsf{as} + * + \mathsf{as}_t \mid (\mathsf{as}, \mathsf{as}_t) \in AS_\alpha(m,n) \times AST_\alpha(m)\} \ ,$$

where $*$ denotes some wildcard characters.

Note that, the number of extended attack strings is

$$|EAS_\alpha(m,n)| = |AS_\alpha(m,n)| \cdot |AST_\alpha(m)| \leq 2^{2m+n} \ .$$

An example extended attack string is $\mathsf{eas} = \mathbf{AH}\cdot\mathbf{A} * \mathbf{A} \in EAS_\alpha(2,1)$. Notice that this eas extends the attack string example of Figure 4. The end of the first epoch in eas allows the adversary to ex-ante fork the chain, while the end of the second epoch provides a selfish mixing opportunity. Intuitively, an $\mathsf{eas} \in EAS_\alpha(m,n)$ string captures the fact that only the beginning and the end of an epoch string can contribute to the manipulative power of a strategic player. Looking ahead, our MDP will have eas strings as its states. We chose $m=6$, $n=2$ to keep our MDP state manageable. We let $\|\mathsf{es}\|_{\mathbf{A}}$ denote the number of adversarial slots in an epoch string $\mathsf{es} \in \{\mathbf{A},\mathbf{H}\}^{32}$.

Handling (2) is more complex, so for now, we focus on evaluating the case presented in Figure 4. For this, it is sufficient to build an MDP for the $EAS(2,1)$ attack strings. In the following subsection, we examine this in detail. This also serves as a small illustrative example involving only 16 states. We can solve the MDP to obtain an optimal strategy, *i.e.*, value iteration converges. As a result, we derive an optimal strategy that essentially consists of only four attack strings ($\mathbf{H}\cdot$, $\mathbf{A}\cdot$, $\mathbf{AA}\cdot$, $\mathbf{AH}\cdot\mathbf{A}$). Surprisingly, for $\alpha \geq 0.2$ this strategy outperforms (by ~ 0.05 more slots) the one based on selfish mixing [1][5], which included 33 attack strings ($\mathbf{A}^t\cdot$, for $t \in [0,32]$).

4.2 A generalised MDP model for $EAS(2,1)$

To illustrate our model of $\mathcal{A}$'s behaviour, we introduce the main concepts using the smallest possible extended attack string space that makes forking attacks possible, *i.e.*, $EAS(2,1)$.

Recall that an MDP is a tuple $M_\alpha = (S, A, \Pi, \{P_\pi\}_\pi, \{R_\pi\}_\pi)$ of states S, action space A, policy $\Pi : S \to A$, transition probabilities $\{P_\pi : S \times S \to \mathbb{R}\}_\pi$ and rewards $\{R_\pi : S \times S \to \mathbb{R}\}_\pi$. If we were to model all possible attack scenarios by an MDP, one would need to create a separate state for each possible point in time where a decision or an observation can be made, and the states should contain information about the observables. This leads to a massive increase in state space, making standard policy iteration techniques infeasible.

Such an explosion of space was also observed in [1], where the following insight is used. Let the state space consist of extended attack strings (number of tail slots), and consider the observables (the possible RANDAO outcomes from various selfish mixing decisions) during state transitions only. Informally, this means that an action (*i.e.*, which slots to miss), the corresponding transition probabilities, and the collected immediate rewards depend not only on the policy and the current state but also on the outcome of an experiment performed in that state. Technically, this is not an MDP any more since applying the same policy and getting the same state transition could potentially lead to different rewards. However, as observed there, this model is still suitable for policy iteration, which yields utility values for states, leading to a profitable policy.

We adopt the strategy above of keeping the state space as small as possible at the price of making state transitions more complicated. Our model consists of a state space of extended attack strings (*cf.* Section 4.2.1), an action space (*cf.* Section 4.2.2) and a policy (*cf.* Section 4.2.3). Note that our policies, as opposed to classical MDPs, yield randomised state transitions, as they result from a "virtual decision tree" corresponding to each state (*cf.* Appendix A.2). We apply a utility function assigning a real number to each state which describes the long-term benefit of being in this state. Immediate rewards and transition probabilities are output as the result of actions (*cf.* Section 4.2.4). In the presentation below, we assume that a utility function $U(s)$ is already available for the extended attack strings s and formalize how one can derive them to the states of the decision trees. In this subsection, we assume that $0.2 \leq \alpha$ to ensure that the forking attack shown in Figure 4 is feasible.

4.2.1 *The simplest MDP's states.*

We have four attack strings, *i.e.*, $AS_\alpha(2,1) := \{\mathbf{H}\cdot, \mathbf{A}\cdot, \mathbf{AA}\cdot, \mathbf{AH}\cdot\mathbf{A}\}$, which we refer to as the honest attack string, two selfish mixing attack strings, and a forking attack string, respectively. The set of possible tail slots is defined as $AST_\alpha(2) := \{\mathbf{H}, \mathbf{A}, \mathbf{AA}, \mathbf{AH}\}$, and we have extended them to length-2 strings for a cleaner notation. Next, we generate $EAS(2,1)$ as in Definition 3 to obtain $4 \cdot 4 = 16$ different extended attack strings, see Figure 6. Let $S := EAS(2,1)$.

4.2.2 *The MDP's action space.*

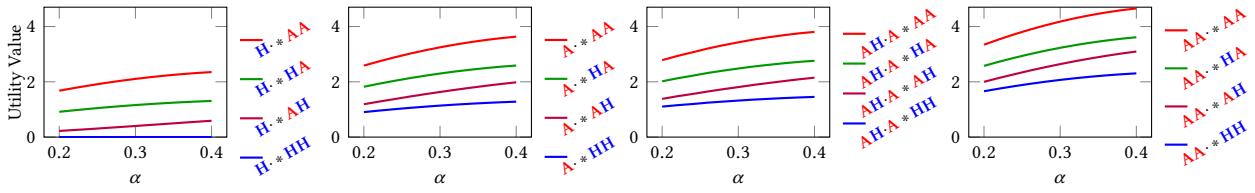
Definition 4 (Attack actions). *An attack action denotes a basic action $\mathcal{A}$ can do. Let $\mathcal{B} := \{\mathbf{Prop}, \mathbf{Miss}, \mathbf{Hide}, \mathbf{Fork}, \mathbf{Regret}\}$, where*

- $\mathbf{Prop}^e_s$ - *$\mathcal{A}$ proposes a block in Slot s.*
- $\mathbf{Miss}^e_s$ - *$\mathcal{A}$ misses in Slot s;* i.e., *does not publish a block.*
- $\mathbf{Hide}^e_s$ - *$\mathcal{A}$ builds a block in Slot s but does not publish it.*
- $\mathbf{Fork}^e_s$ - *$\mathcal{A}$ builds a block in Slot s on the last hidden block.*
- $\mathbf{Regret}^e_s$ - *$\mathcal{A}$ "forgets" its private blocks, foregoes forking.*

A more detailed description of each action can be found in Algorithm 2. Subscripts are omitted whenever they are clear from the context.

Let $A := \mathcal{B}^*$. To illustrate the previously introduced attack actions on the decision trees shown in Section 3.2. In case of a **selfish mixing** attack string ($\mathbf{A}^t, t \in \{1,\dots,32\}$), *cf.* Figure 3, each 2^t out-edges correspond to the $\{\mathbf{Prop}, \mathbf{Miss}\}^t$ actions. Forking attack

[5] The term "optimal" refers to a specific set of attack strings.



(a) Attack strings with prefix H· (b) Attack strings with prefix A· (c) Attack strings with prefix AH·A (d) Attack strings with prefix AA·

Figure 6: The utility function for each of the 16 extended attack string of $EAS(2,1)$.

strings admit a richer action space. For the sake of concreteness and simplicity consider the forking eas = $\mathbf{AH{\cdot}A}$. $\mathcal{A}$ has two decisions to make. The first decision is before $\mathcal{A}$ sees the RANDAO contribution r^e_{31}, and the second decision is after the publication of $\mathbf{H}^{31}_e$.

- **Prop**$^e_{30}$: $\mathcal{A}$ behaves honestly, *i.e.*, $\mathcal{A}$ publishes the block $\mathbf{A}^e_{30}$.
- **Hide**$^e_{30}$: $\mathcal{A}$ starts forking by privately building $\mathbf{A}^e_{30}$. After r^e_{31} is published, $\mathcal{A}$ makes another decision:
  - **Fork**$^{e+1}_0$: completes forking by proposing $\mathbf{A}^{e+1}_0$ upon $\mathbf{A}^e_{30}$.
  - **Regret**$^{e+1}_0$**Prop**$^{e+1}_0$: the so-called "regret" branch of the decision tree. $\mathcal{A}$ foregoes forking by proposing $\mathbf{A}^{e+1}_0$ on top of $\mathbf{H}^e_{31}$, essentially forfeiting $\mathbf{A}^e_{30}$.

For a more detailed description of $\mathcal{A}$'s action, see Algorithm 2.

4.2.3 The MDP's policy. We assign a decision tree to each state to describe a policy, which we do in Appendix A.3. $\mathcal{A}$ in epoch e constructs a decision tree based on eas$_e$, walks through this tree (*e.g.*, Figures 4 and 5), which yields R^e and the next state eas$_{e+1}$.

At a high level, we briefly describe now the structure of the decision trees corresponding to forking attack strings (starting with $\mathbf{A}^{a_1}\mathbf{H}^h\mathbf{A}^{a_2}$). $\mathcal{A}$ decides to behave honestly based on a threshold (*cf.* Definition 6) or starts forking. If $\mathcal{A}$ decides to start forking (*e.g.*, by privately building all a_1 blocks: **Hide**^{a_1}), after $\mathbf{H}^h$, $\mathcal{A}$ can finish ex-ante reorging or "regret" the attack by abandoning $\mathbf{A}^{a_1}$. An important observation is that, in both cases, after a successful forking or regretted attack, $\mathcal{A}$ still aims to maximise the utility until the epoch boundary, *regardless of the sacrificed previous blocks*. This recursive structure lets us describe the new state with a shorter attack string from the remaining slots in epoch e, *cf.* Section 4.3. This observation holds for the case of "no forking" as well, since after proposing $\mathbf{A}^{a_1}\mathbf{H}^h$, $\mathcal{A}$ still seeks to maximise the utility. We define (the multisets of) observations to model each decision situation. Intuitively, whenever $\mathcal{A}$ could manipulate the RANDAO, it observes and arbitrarily chooses from different RANDAO outputs $R^{e,*}$ along with their corresponding extended attack strings eas, actions act leading towards $R^{e,*}$ and the number of sacrificed blocks sac. However, for the action of behaving honestly, there are no observations, the expected utility is compared to the derived utilities of the observations.

Definition 5. *Let* $O := \{(\mathsf{c}, \mathsf{act}, \mathsf{eas}, sac) \in \{\mathbf{C}, \mathbf{N}\}^m \times \mathcal{B}^* \times EAS(m,n) \times \mathbb{N}\}$ *be the observation set, where* c *is the realisation string yielding* eas *and* sac*, while* act *is the corresponding action towards* c. Ω *denotes the set of all possible multisets of observations.*

We compute the immediate reward by evaluating $cnt(\mathsf{eas}, sac) := \|\mathsf{eas}[-32:]\|_{\mathbf{A}} - sac$ by choosing a certain eas. A policy Π is defined as a total ordering on Ω and a threshold corresponding to the "no forking" action, *cf.* Definition 6. For selfish mixing states s with t tail slots, one can evaluate the utility $U(s) = \max_{i\in[1,2^t]}(cnt(\mathsf{eas}_i, \mathsf{sac}_i) + U(\mathsf{eas}_i))$ where each $(\mathsf{eas}_i, \mathsf{sac}_i)$ corresponds to the i^{th} observation. There are different actions $\mathcal{A}$ can perform: it can either decide not to build a private chain, referred to as the "no forking" action, or it can choose to build a private chain, which can be done in multiple ways. Next, we define utility thresholds based on which $\mathcal{A}$ decides which action to choose.

Definition 6 (No forking threshold). *For any MDP state* $s \in S$*, we define (*cf. *[25, Eq.(17.4)]) a threshold for the "no forking" action* $a \in A(s)$ *(the action* a *is defined by the corresponding* eas *forking string):* $\mathsf{N}_\alpha(s) := \sum_{s'\in S} P(s'|s,a)U(s')$*. Here,* s' *is the states reached after the "no forking" action,* i.e.*,* $\mathcal{A}$ *proposes a block honestly in all its assigned slots* $\mathbf{A}^{a_1}$*. Note that this threshold does not depend on the observations of* s*.*

Definition 7 (Forking threshold). *For a given MDP state* s*, and forking action* $a \in A(s)$*,* $\mathcal{A}$ *can observe a set of observables* $O \in \Omega$*. The forking threshold is defined as* $\mathsf{F}_\alpha(s,a,O) := \sum_{s'\in S_O} P(s'|s,a)U(s')$*. Here,* S_O *is the set of states in* O*,* i.e.*, the observable states in the decision tree after the forking action* a*.*

Theorem 3. *For any forking* eas *of a state* s *and observations* $O \in \Omega$*, the following equation holds for the optimal utility:*

$$U(s) = \max(\{\mathsf{N}_\alpha(s), \mathsf{F}_\alpha(s,a_1,O), \ldots, \mathsf{F}_\alpha(s,a_n,O)\}) \ , \qquad (5)$$

where $\{a_1,\ldots,a_n\} \subset A(s)$ *are the actions where* $\mathcal{A}$ *starts ex-ante reorging in a specific way.*

The proof is deferred to Appendix A.1.

Naturally, $\mathcal{A}$ chooses the action with the largest utility. We already discussed the utility corresponding to ex-ante reorging, but once $\mathcal{A}$ started it by hiding some of its $\mathbf{A}^{a_1}$ blocks, after $\mathbf{H}^h$, $\mathcal{A}$ can regret forking. From the new observations $\mathcal{A}$ can decide to do so accordingly in the shorter eas$_e{}'$. Recall, that we could describe a new state still in epoch e after a regretted or successful forking with the remaining slots, adjusting with the already sacrificed blocks.

4.2.4 The MDP's transition probability and reward functions. The following functions describes the state transition between $EAS_\alpha(m,n)$.

Definition 8 (The next(·) function). *The function* $\mathsf{next}^\Pi_{R,(m,n)} : EAS_\alpha(m,n) \to \mathbb{Z} \times EAS_\alpha(m,n)$ *gets* eas$_e$*, the current RANDAO state* $R \in \{0,1\}^{256}$ *and produces the immediate reward* cnt *and the next* eas$_{e+1}$ *according to* Π*.*

In the following, we do not denote the parameters Π, R and (m, n) of next$(\cdot)$, as they will be obvious from the context.

One can compute an optimal MDP with value iterations solely on $EAS_\alpha(m, n)$. Two following equation formally gives the required steps after an initialising U_0 (*e.g.*, $U_0 \equiv 0$):

$$U_{i+1}(\mathsf{eas}_e) = \mathbb{E}_R(cnt + U_i(\mathsf{eas}_{e+1})) \ , \tag{6}$$

where $(cnt, \mathsf{eas}_{e+1}) = \mathsf{next}(\mathsf{eas}_e)$.

Definition 9 (Stochastic Observations). *An observation corresponding to a realisation string* c *and to the current state described by* eas_e *yields a sacrifice (constant) and* eas_{e+1}. *The epoch string of epoch* $e+2$: $\mathsf{eas}_{e+1}[-32:]$ *consists of 32 independent Bernoulli trials, where each trial yields* A *with probability* α *and* H *with probability* $1-\alpha$. *We denote this distribution as* $X_{\mathsf{eas},\mathsf{c}}$ $(\mathsf{c} \in \{\mathbf{C}, \mathbf{N}\}^m)$.

Recall that we assume U is given on $EAS_\alpha(2, 1)$.

Honest attack strings. Here, $\mathcal{A}$ cannot manipulate the RANDAO. Thus, O is a singleton set; $O := \{(\epsilon, \epsilon, \mathsf{eas}_1, \mathsf{sac}_1)\}$. The new state is $\mathsf{eas} := \mathsf{eas}_1$ and *rewards* += $cnt(\mathsf{eas}_1, \mathsf{sac}_1)$.

Selfish mixing eas**.** Let t denote the adversarial tail length in $\mathsf{eas}[:-32]$. Then, $O = \{(\mathsf{c}_i, \mathsf{act}_i, \mathsf{eas}_i, \mathsf{sac}_i)_{i=1}^{2^t}\}$. $\mathcal{A}$ chooses the observation $\omega_j = (\mathsf{c}_j, \mathsf{act}_j, \mathsf{eas}_j, \mathsf{sac}_j) \in O$ according to $U(\cdot)$. The new state $\mathsf{eas} := \mathsf{eas}_j$ and *rewards* += $cnt(\mathsf{eas}_j, \mathsf{sac}_j)$.

Forking attack strings. In a forking state s, which corresponds to eas_s starting with attack string **AH·A**. $\mathcal{A}$ can only observe $\omega_f = (\mathbf{CN}, \mathbf{Hide}, \mathsf{eas}_f, 0) \leftarrow X_{\mathsf{eas}_s,\mathbf{CN}}$ according to which $\mathcal{A}$ decides to fork or not. If $\mathcal{A}$ decides not to fork, the remaining attack string will be "·**A**" after proposing **AH**, thus $\mathcal{A}$ would observe $(\cdot, \cdot, \mathsf{eas}_n, 0) \sim X_{\mathsf{eas}_s,\mathbf{C}^2}$. The threshold of this action is $\mathsf{N}_\alpha(s) = \mathbb{E}(cnt(\mathsf{eas}_n, 0) + U(\mathsf{eas}_n))$. Whether $\mathcal{A}$ starts forking also depends on the threshold $\mathsf{F}_\alpha(s, \mathbf{Hide}, \{\omega_f\})$. Let $u_f := cnt(\mathsf{eas}_f, 0) + U(\mathsf{eas}_f)$ denote the utility of the state of completing the ex-ante reorg, thus reaching the realisation string **CN**. Recall that $\mathcal{A}$ may regret forking, leading to eas_r, where $(\cdot, \cdot, \mathsf{eas}_r, 1) \sim X_{\mathsf{eas}_s,\mathbf{NC}}$, which $\mathcal{A}$ can only sample in Slot 30. The random variable of its utility is $u_r := cnt(\mathsf{eas}_r, 1) + U(\mathsf{eas}_r)$, which $\mathcal{A}$ compares to u_f in the first slot of the next epoch. Finally, the forking threshold can be computed as $\mathsf{F}_\alpha(\mathsf{eas}_e, \mathbf{Hide}, \{\omega_f\}) = \mathbb{E}(\max(u_f, u_r))$.

4.3 Forking attacks' recursive characterization

The advantage of the $EAS(2, 1)$ example was that it included only a single forking attack. As shown in Figure 5, many more such attack strings exist, and they can be recursively reduced to one another. Recall that Figure 5 extended the example of Figure 4 by including the possibility of selfish mixing attacks on two branches. In our evaluation, we employed a recursive construction to characterise these attacks. Next, we highlight the key ideas; the details of our recursive approach are deferred to Appendices A.2 and A.3.

The following examples illustrate some of the intricacies of forking attack strings that make their analysis challenging.

Self-forking at the head slot (AHA·A) The tail-forking attack can be even more profitable if the adversary also controls the next head slot. In such cases – surprisingly – the decision tree for $\alpha > 0.32$ includes a branch where it may be more profitable for the adversary to miss its own block $\mathbf{A}_{31}$. Put differently, if $\alpha > 0.32$, then $\{\mathsf{P}^e_{29}, \mathsf{R}^e_{30}, \mathsf{M}^e_{31}, \mathsf{H}^{e+1}_0\}$ is a viable configuration. $\mathcal{A}$ may be incentivised to prolong the forking attack and *miss its own block* $\mathbf{A}_{31}$, akin to the sacrificed blocks in selfish mixing.

Different decision tree for tail forking depending on α ($\mathbf{A}^2\mathbf{HA}\cdot$) If $0.15 \le \alpha < 0.2$, then $\mathcal{A}$ must start building a branch from $\mathbf{A}_{28}$ preceding $\mathbf{H}_{30}$ for forking. $\mathcal{A}$ is able to fork out $\mathbf{H}_{30}$, while selfish mixing at the same time. Consider the two possible slot statuses $\{\mathsf{P}^e_{28}, \mathsf{P}^e_{29}\mathsf{R}^e_{30}\mathsf{H}^e_{31}\}$ and $\{\mathsf{P}^e_{28}, \mathsf{M}^e_{29}, \mathsf{R}^e_{30}, \mathsf{H}^e_{31}\}$. In other words, $\mathcal{A}$ can fork out $\mathbf{H}_{30}$ even sacrificing $\mathbf{A}_{29}$. On the other hand, if $0.2 \le \alpha$, $\mathcal{A}$ can minimize the sacrifice of a regretted fork while reaching the same $R^{e,*}$ with $\{\mathsf{H}^e_{28}, \mathsf{P}^e_{29}, \mathsf{R}^e_{30}, \mathsf{H}^e_{31}\}$ by proposing $\mathbf{A}_{28}$. This way, when $\mathcal{A}$ builds on top of $\mathbf{A}_{28}$, only sacrificing $\mathbf{A}_{29}$. Finally, $\mathcal{A}$ can fork out $\mathbf{H}_{30}$, while missing $\mathbf{A}_{28}$, because $\mathcal{A}$ can gather enough votes even if the forking starts forking in Slot 28. A generalization of how much slot is required before **H** is stated in Theorem 1.

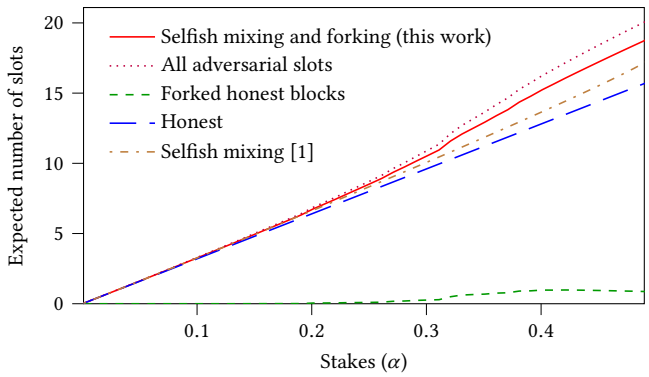


Figure 7: Efficacy of various RANDAO manipulation strategies. Note that a strategic player who applies both forking and selfish mixing strategies outperforms the RANDAO manipulator using only selfish mixing. As a result of RANDAO manipulation, chain quality is also reduced, i.e., see the increased number of forked honest blocks.

Recursive (forking) attack strings As Theorem 2 suggests attack strings can be built recursively. We illustrate this recursion with the attack string as = **AHAAHA**·. Each action $\mathcal{A}$ performs, reduces the attack string to a shorter as, we already evaluated.

- **Prop**$_{26}$ yields $\mathbf{A}^2\mathbf{HA}\cdot$ (since $\mathbf{H}_{27}$ is always proposed). Thus the realisation string$(\in \{\mathbf{C}, \mathbf{N}\}^6)$ it will start with $\mathbf{C}_{26}\mathbf{C}_{27}$.
- **Hide**$_{26}$ action produces the following possibilities:
  - **Fork**$_{28}$: realisation string will start with $\mathbf{C}_{26}\mathbf{N}_{27}\mathbf{C}_{28}$ and $\mathcal{A}$ has to maximise the utility in **AHA**.
  - **Regret**$_{28}$: in this case, the realisation string starts with $\mathsf{c} = \mathbf{N}_{26}\mathbf{C}_{27}$. Afterwards, $\mathcal{A}$ maximises the utility in the attack string $\mathbf{A}^2\mathbf{HA}\cdot$ regardless of the lost block $\mathbf{A}_{26}$.

4.4 Results

We solved our generalised MDP for the extended attack strings described in Appendix A.2, *i.e.*, for $EAS_\alpha(6, 2)$. We observe that RANDAO manipulations with forking strategies already outperform selfish mixing with $\alpha = 0.08$, *cf.* Figure 7. A RANDAO manipulator staker with $\alpha = 41.95\%$ staking power could obtain 50%

of the proposed slots. In contrast, selfish mixing alone would require an $\alpha = 46.24\%$ to get half of the slots. We see in Figure 12 that a RANDAO manipulator validator significantly decreases the blockchain throughput by missing blocks in its own slots and by forking out honest blocks. For example, $\mathcal{A}$ with $\alpha = 0.33\%$ decreases the blockchain's throughput by 3.86%. The decreased transaction throughput is a negative externality for every user of the system. In Figure 8, we analyze the probability distribution of the different actions in our generalised MDP's stationary distribution. For example, for $\alpha = 0.30$, we see that a strategic $\mathcal{A}$ performs an ex-ante reorg in 18.94% of the epochs, regrets forking in 3.31% of the epochs, selfish mixes in 15.99% of the epochs and only acts honestly in 61.76% of the epochs. In actuality, a blockchain network using an *unbiased randomness beacon* should only incentivise honest behaviour.

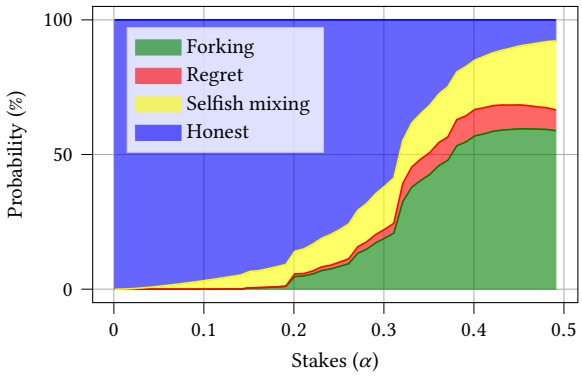


Figure 8: The probability of each strategic behaviour in the stationary distribution. If $\mathcal{A}$ ex-ante reorged once at the epoch, it counts as Forking, if regretted an attack but did not fork, it counts to to the Regret category etc. Large staking entities are often incentivised to manipulate the RANDAO. For instance, an economically rational staker with $\alpha = 0.3$ manipulates the RANDAO in 38.24% of epochs.

Table 2 compares strategic players' manipulative power in PoS Ethereum and PoW Bitcoin using selfish mixing with/without forking and selfish mining. Observe that the corresponding effective stakes with the forking strategy surpass that of selfish mixing alone when the attacker has large stakes, and it can efficiently fork out honest blocks. Currently, the largest staking pool controls 28.1%[6] of the stakes. This staking power can be increased by 8.37% with our combination of block withholding and forking attacks. By comparing the numbers in Table 2 we can say that the current DRB in the Ethereum network is less biasable than that of a PoW blockchain.

5 RANDAO MANIPULATIONS IN PRACTICE: EMPIRICAL MEASUREMENTS

This section focuses on whether any RANDAO manipulation attacks have occurred. We present our measurements, which were performed to find empirical evidence of such attacks. Primarily, we examined those epoch boundaries where there are missed slots at the beginning or end of an epoch. To effectively investigate these attacks, we need to group validators and assume that the attacker is part of one or more of these groups. Each block proposer can be identified by their public key. Note that this data alone is insufficient to perform statistical tests because a staking entity typically controls numerous validators. Luckily, some entities reveal their public keys, and a mapping between proposers and entity identities can be constructed. To that end, we use public data obtained using various APIs, [7] and process the blocks over a two-year period, from September 22, 2022, to October 1, 2024 (epochs in the range [148412, 314998]), [8] extracting the actual block proposers and the number of missed blocks for each epoch.

Stake (α)	**Ethereum PoS** Selfish Mixing (SM) [1]	**Ethereum PoS** This work SM+forking	**Bitcoin PoW** Selfish Mining [12, $\gamma = 1$ in (3)]
1%	1.0011%	1.0011%	1.0099%
5%	5.0483%	5.0483%	5.2387%
10%	10.1881%	10.1882%	10.9194%
15%	15.3996%	15.4243%	17.0110%
20%	20.6777%	20.9583%	23.5165%
25%	26.0247%	26.6090%	30.4878%
30%	31.4516%	32.8356%	38.0622%
35%	36.9735%	40.2870%	46.5560%
40%	42.6244%	47.4946%	56.7442%
45%	48.4918%	53.7777%	70.9423%

Table 2: Advantages in terms of staking power of strategic RANDAO manipulators. We compare this work and prior work by Alpturer and Weinberg [1] for various stake sizes.

5.1 Selfish mixing

First, we search for statistical evidence of systematic RANDAO manipulations using selfish mixing on the Ethereum mainnet.

5.1.1 Missing consecutive tail slots. Major staking entities have had numerous opportunities to conduct selfish mixing RANDAO manipulations, see Figure 9. We found that the biggest staker, Lido, could have manipulated the RANDAO with selfish mixing 47 694 times. In particular, Lido had 4 consecutive tail slots in 737 different epochs and missed at least one slot in 3 of these instances.

5.1.2 Not missing tail slots. We focus on consecutive tail slots, where entities did not miss any tail slots. Recall that in this case, all counterfactual RANDAO outputs $R^{e,*}$ in epoch e are computable by anyone, cf. Section 3.4.1. Given t tail slots, the adversary could have chosen from 2^t RANDAO outputs. We label the realised R^e as **BEST** if not missing any slot was a better strategy for maximizing the reward $(p(e+2, R^e, \mathcal{A}) - s(e, R^e, \mathcal{A}))$. We call a RANDAO output **NEUTRAL** for the tail slot owner if there were other achievable configurations (*i.e.*, by missing tail slots) that would have yielded the same number of slots. Finally, the RANDAO output is labelled as **WORSE** if the adversary could have chosen a configuration

[6] See: https://beaconcha.in/pools.

[7] https://beaconcha.in/, https://beaconscan.com/, https://etherscan.io/.

[8] Before the merge (September 2022) the beacon chain and the RANDAO had not been used to select validators. Hence, we disregard epochs $e \in [0, 148411]$.

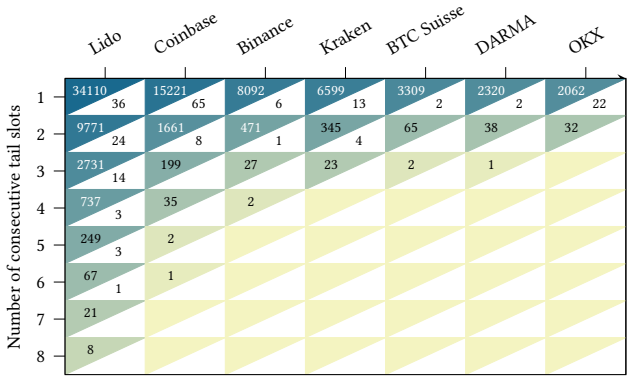


Figure 9: Selfish mixing manipulation opportunities for major staking entities in epochs [148412 − 314998]**. The values in the coloured triangles indicate the number of epochs during which the staking entity had a certain number of consecutive tail slots, while the white triangles indicate how many of these epochs included at least one missed tail slot.**

that would have yielded strictly more slots in epoch $e + 2$ than the one realised by choosing not to miss any slots. As illustrated in

Table 3: Selfish mixing strategies when stakers did not miss any tail slots. Note that entities' staking power typically changes over time. Here, α denotes the entities' average staking power in the samples of selfish mixing candidates.

Entity	BEST	NEUTRAL	WORSE	Stake* (α)
Lido	14 788	3 244	12 883	28.5
Coinbase	9 224	2 685	4 954	10.96
Binance	5 151	1 616	1 745	5.63
Kraken	4 318	1 319	1 268	5.34
Bitcoin Suisse	2 359	701	290	2.06
Upbit	1 184	253	53	1.05
OKX	1 540	399	120	1.48

Table 3, in several cases, major staking entities might have achieved better outcomes by withholding the publication of all their tail slots. For instance, an economically rational Lido operator should have missed at least one slot 12 883 times when, in fact, they did not miss.

We examined the distribution of slots for the major staking entities when their validators proposed all tail slots. Let N_p be the number of such epochs and $1 \le k \le N_p : e_k^p$ be the k-th such epoch. If no RANDAO manipulation occurred with the utility of maximizing the number of proposed blocks, the number of obtained slots $p(e+2, \cdot, \mathcal{A})$ in epoch $e + 2$ by an entity would be binomially distributed see Figure 10

$$H_0 : \forall k \in \mathbb{N}(1 \le k \le N_p) : p(e_k^p, \cdot, \mathcal{A}) \sim \text{Binom}(32, \alpha_{e_k^p}) \ . \quad (7)$$

To statistically test whether $\forall k \in [1, N_p] : p(e_k^p, \cdot, \mathcal{A})$ are from the established distributions, we take the histogram of the $p(e_k^p, \cdot, \mathcal{A})$

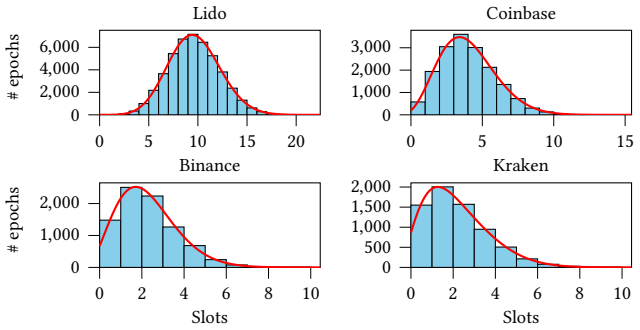


Figure 10: Number of slots in epoch $e + 2$ without missed slots in epoch e for Lido, Coinbase, Binance, and Kraken. The blue bars represent the empirical distribution, while the red line is the expected theoretical distribution for the number of obtained slots in epoch $e + 2$.

values between 0 and 32. We define an indicator variable as $I_{k,i} = 1$ if $p(e_k^p, \cdot, \mathcal{A}) = i$, and 0 otherwise. Let H^p be the empirical distribution of $p(e^p, \cdot, \mathcal{A})$, where $H_i^p = \sum_{k=1}^{N_p} I_{k,i}$. If H_0 holds, $I_{k,i} \sim \text{Bernoulli}\left(\binom{32}{i}\alpha_{e_k^p}^i(1-\alpha_{e_k^p})^{32-i}\right)$, thus

$$\hat{H^p} := \mathbb{E}(H_i^p) = \sum_{k=1}^{N_p}\mathbb{E}(I_{k,i}) = \sum_{k=1}^{N_p}\binom{32}{i}\alpha_{e_k^p}^i(1-\alpha_{e_k^p})^{32-i} \ . \quad (8)$$

We can test whether H^p matches $\hat{H^p}$ with the χ^2 test. We grouped every adjacent bin in the histogram, so for $i \in 0 \le i \le$ **Bins** : $H_i^{'} \ge 5$, as the chi-squared test recommends. According to the results, see Table 4, no p**-value** takes a critically low value.

5.1.3 Missing a few tail slots. Selfish mixing entails selectively missing tail slots, but it can also caused by other reasons, *e.g.*, network outage. Next, we test this hypothesis. The null hypothesis states that all the times an entity missed at the end of the epoch, it was due to unrelated reasons to the RANDAO. In other words, rejecting the null hypothesis indicates that systematic RANDAO manipulation must have been present. Generally, considering the epochs in which an entity has missed at least one slot, the number of slots obtained $p(e, \cdot, \mathcal{A})$ should be distributed as the sum of discrete binomial distributions, a Poisson binomial distribution. Let N be the number of epochs in which the entity in question missed at least one tail slot, and for $1 \le k \le N_m$, let e_k^m represent the k-th such epoch. Note that α_e, the staking power of the validator entity in question, changes over time. Thus, we cannot use standard distribution statistical tests, *e.g.*, the Z-test. Instead, we test whether

$$\sum_{k=1}^{N_m} p(e_k^m, \cdot, \mathcal{A}) \sim \text{PoissonBinom}(32, (\alpha_{e_1^m}, \alpha_{e_2^m} \ldots, \alpha_{e_{N_m}})) \ . \quad (9)$$

If $N_m > 50 \wedge \forall k \in [1, N] : \alpha_{e_k^m} > 0.05$, we can approximate this distribution with a normal distribution. We perform a right-tailed normality testing on the value $\sum_{k=1}^{N_m} p(e_k^m, \cdot, \mathcal{A}) \sim \mathcal{N}(\mu, \sigma^2)$. Let $\mu_0 = 32\sum_{i=1}^{N}\alpha_{e_i}$ and $\sigma_0 = \sqrt{32\sum_{i=1}^{N}\alpha_{e_i}(1-\alpha_{e_i})}$. The null hypothesis is $H_0 : \mu \le \mu_0$, while the alternative hypothesis is $H_1 : \mu > \mu_0$. We cannot reject the null hypothesis in all applicable cases due to the

large p-values, see Table 5. Thus, missed tail slots are not part of a systemic RANDAO manipulation by any of the studied entities.

Table 4: χ^2 test on the number of slots entities obtained when proposed all tail slots. N denotes the number of examined epochs and *Bins* the bins of the histogram.

Validator	N	Bins	p-value
Lido	46 879	19	0.852
Coinbase	16 883	13	0.325
Binance	8 512	8	0.142
Kraken	6, 905	9	0.803
Bitcoin Suisse	3 350	6	**0.055**
Upbit	1 490	4	0.465
OKX	2 059	5	0.303

Table 5: Z-test results applied to the number of slots obtained by Lido and Coinbase in epochs where they missed at least a tail slot. No evidence for systematic RANDAO manipulations.

Validator	N	μ	μ_0	σ_0	p-value
Lido	80	746	752.21	23.039	0.606
Coinbase	71	222	228.344	14.304	0.671

5.2 Forking RANDAO manipulations

Next, we search for statistical evidence of systematic RANDAO manipulations using ex-ante and ex-post forking.

5.2.1 Forking for slot number maximizing utility. With current staking powers, only a handful of entities can execute an ex-ante reorg. Possible candidates include Lido, Coinbase, Kraken, etc. The difficulty in reorging is not only the presence of enough staking power but also the right tail slot configuration, *e.g.*, **A**H**A**·. We only found two instances at epochs 192 420 and 313 291 where Lido ex-ante reorged Gemini and a solo staker at the 30^{th} slot of the epoch, respectively. Lido obtained more slots (8 in both cases) in the next but one epoch. If Lido had not forked out them and had behaved honestly, then Lido would have obtained only 7 slots in epoch 192 422 and 313 291. We empirically evaluated Lido's unrealised rewards. In particular, Lido could have manipulated the RANDAO numerous times, even though it did not engage in these manipulations. This economically irrational behaviour resulted in 74 782 slots of unrealised rewards between epochs 190 000 and 314 994.

5.2.2 Forking and MEV. Many works analyze and quantify the incentives implied by MEV [10, 18, 23, 30]. Next, we study the potential correlation (or lack thereof) between the incentives due to MEV and RANDAO manipulation. Specifically, we considered instances of ex-ante reorgs and ex-post reorgs (cf. Section 3.3) that happened for the strings ending with H·**A** or H**A**·. Recall ex-post reorgs can occur in practice accidentally if **H** had been published late into its slot and received an unusually low number of votes. We observe many reorgs executed by three major staking entities. However, we see no correlation between RANDAO and MEV incentives, see Appendix C.2 for our empirical measurements.

6 COUNTERMEASURES

An often-mentioned countermeasure for RANDAO manipulations in the Ethereum community is social slashing [29]. There are several severe challenges to this approach. First, social slashing is hard to agree on and enforce in a decentralised environment. Second, social slashing may become an even larger centralizing force than it attempts to prevent, in this case, the unfair enrichment of already wealthy validators. Third, algorithmic countermeasures are desired instead of social deterrents in a trust-minimised, geo-economically decentralised environment. On the other hand, there seem to be no simple, immediate algorithmic or cryptographic countermeasures without fundamentally redesigning the currently used RANDAO protocol. Alpturer and Weinberg observed that the manipulability of the RANDAO is attenuated with longer epochs because longer epochs' tail slots yield less manipulative power [1]. Additionally, the protocol could financially penalise not publishing blocks. If the proposer boost parameter were decreased, ex-ante reorgs would be harder to accomplish. However, note that none of these countermeasures eliminates RANDAO manipulability *entirely*.

We highlight two cryptographic protocols that could offer a viable, *unbiasable* alternative to the current RANDAO protocol. First, verifiable delay functions (VDF) eliminate the possibility of bias by delaying the DRB's output after all contributions have been made [4]. If the delay parameter is chosen properly, no bias is possible in the *dishonest majority setting* (under cryptographic assumptions). VDFs are non-trivial to deploy [3] and have severe hardware requirements. We also consider weighted threshold VRFs [11] as a promising cryptographic primitive for instantiating an unbiasable DRB in the *honest majority setting*. Unfortunately, the currently known only BLS-based weighted threshold VRF protocol [11] does not scale to millions of validators, the scale Ethereum would need.

7 RELATED WORK

The manipulability of leader selection mechanisms in Byzantine fault-tolerant PoS blockchains has been analysed in [13, 14] to uncover the relationship between the power of the adversary and the network connectivity parameter. Yaish et al. proceed by defining multiple variants of a timestamping attack and find that these attacks were performed in the wild [32], making it the first confirmed case of consensus-level manipulation in a major cryptocurrency.

A great summary of DRBs can be found in [9, 19, 24]. The vulnerabilities of the DRB in PoS Ethereum have been discussed by the community since 2018 [6, 21, 28, 29]. Then, an initial formal verification analysis of an earlier, two-round version of the RANDAO [33] protocol was presented in [2]. Alpturer and Weinberg [1] show that the RANDAO manipulation game in Ethereum can be formulated as an MDP and propose a state-space reduction method that allows policy iteration to converge quickly on a laptop.

8 CONCLUSION AND FUTURE DIRECTIONS

We presented the currently known most powerful RANDAO manipulations by considering the selfish mixing strategy and also

the ex-ante forking strategy, where the adversary forks out honest blocks from the canonical chain to increase its manipulative power.

We foresee three main areas for future work. A complete, albeit technically much more difficult, MDP could model even more strategies for possible RANDAO manipulation. In our current model, we do not consider slot-varying rewards (i.e., slots' values are not uniform), neither ex-post reorgs nor the manipulation and suppression of honest attestations in sync committees.

We foresee the emergence of a RANDAO bribery market, where validators can auction off their RANDAO manipulation rights if they obtain tail slot(s) in an epoch. A corrupt validator could publish their own RANDAO contribution before their allotted tail slot(s). After learning every RANDAO reveal, market participants could offer bribes to the corrupt validator not to publish their tail slot(s). Note that such a market could be implemented trustlessly using smart contracts. The game-theoretic implications of such a complex market could be studied in future work.

Future work should amend (*e.g.*, longer epochs), fix or even replace the current RANDAO protocol with an adequate non-biasable DRB protocol (*e.g.*, VDFs [4], weighted threshold VRFs [11]), given the Ethereum consensus protocol's stringent latency requirements.

ACKNOWLEDGMENT

We are grateful to Joachim Neu for insightful discussions on various attacks on Ethereum. We thank Arantxa Zapico and Gottfried Herold for discussions on RANDAO bribery markets. We are indebted to Kaya Ito Alpturer and Matt Weinberg for sharing their manuscript on optimal RANDAO manipulations with us. Last but not least, we thank Toni Wahrstätter for inspiration. István András Seres was supported by the Ministry of Culture and Innovation and the National Research, Development, and Innovation Office within the Quantum Information National Laboratory of Hungary (Grant No. 2022-2.1.1-NL-2022-00004). Bence Ladóczki and János Tapolcai received financial support from the National Research, Development, and Innovation Office (NKFIH, Grant No. K-146347).

REFERENCES

[1] Kaya Alpturer and Matthew Weinberg. 2024. Optimal RANDAO Manipulation in Ethereum. *Advances in Financial Technologies* (2024).

[2] Musab A Alturki and Grigore Roşu. 2020. Statistical model checking of RANDAO's resilience to pre-computed reveal strategies. In *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part I 3*. Springer, 337–349.

[3] Alex Biryukov, Ben Fisch, Gottfried Herold, Dmitry Khovratovich, Gaëtan Leurent, María Naya-Plasencia, and Benjamin Wesolowski. 2024. Cryptanalysis of algebraic verifiable delay functions. In *Annual International Cryptology Conference*. Springer, 457–490.

[4] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. 2018. Verifiable delay functions. In *Annual international cryptology conference*. Springer, 757–788.

[5] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International conference on the theory and application of cryptology and information security*. Springer, 514–532.

[6] Vitalik Buterin. 2018. RANDAO beacon exploitability analysis, round 2. https://ethresear.ch/t/randao-beacon-exploitability-analysis-round-2/1980. Accessed: 2024-11-12.

[7] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and casper. *arXiv preprint arXiv:2003.03052* (2020).

[8] Miles Carlsten, Harry Kalodner, S Matthew Weinberg, and Arvind Narayanan. 2016. On the instability of bitcoin without the block reward. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 154–167.

[9] Kevin Choi, Aathira Manoj, and Joseph Bonneau. 2023. SoK: Distributed Randomness Beacons. Cryptology ePrint Archive, Paper 2023/728. https://eprint.iacr.org/2023/728

[10] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2019. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. arXiv:1904.05234 [cs.CR]

[11] Sourav Das, Benny Pinkas, Alin Tomescu, and Zhuolun Xiang. 2024. Distributed randomness using weighted vrfs. *Cryptology ePrint Archive* (2024).

[12] Ittay Eyal and Emin Gün Sirer. 2014. Majority is not enough: Bitcoin mining is vulnerable. In *Int. Conf. on Financial Cryptography and Data Security (FC)*. Springer, 436–454.

[13] Matheus VX Ferreira, Ye Lin Sally Hahn, S Matthew Weinberg, and Catherine Yu. 2022. Optimal strategic mining against cryptographic self-selection in proof-of-stake. In *Proceedings of the 23rd ACM Conference on Economics and Computation*. 89–114.

[14] Matheus V. X. Ferreira, Aadityan Ganesh, Jack Hourigan, Hannah Huh, S. Matthew Weinberg, and Catherine Yu. 2024. Computing Optimal Manipulations in Cryptographic Self-Selection Proof-of-Stake Protocols. In *The 25th ACM Conference on Economics and Computation (EC '24)* (New Haven, CT, USA, July 8–11, 2024). ACM, New York, NY, USA, 43. https://doi.org/10.1145/3670865.3673602

[15] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[16] Ethereum Foundation. 2023. Ethereum Proof-of-stake documentation. https://github.com/ethereum/consensus-specs/blob/8696fbf75387fb37a32fc08a6b934653198c6c0c/specs/phase0/validator.md#attesting. Accessed: 2024-11-12.

[17] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. 2012. An enciphering scheme based on a card shuffle. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. Springer, 1–13.

[18] Aljosha Judmayer, Nicholas Stifter, Philipp Schindler, and Edgar Weippl. 2022. Estimating (miner) extractable value is hard, let's go shopping!. In *International Conference on Financial Cryptography and Data Security*. Springer, 74–92.

[19] Alireza Kavousi, Zhipeng Wang, and Philipp Jovanovic. 2024. SoK: Public Randomness. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. 216–234. https://doi.org/10.1109/EuroSP60621.2024.00020

[20] Andrew Lewis-Pye and Tim Roughgarden. 2023. Permissionless Consensus. *arXiv preprint arXiv:2304.14701* (2023).

[21] Paul D. 2018. Limiting last-revealer attacks in beacon chain randomness. https://ethresear.ch/t/limiting-last-revealer-attacks-in-beacon-chain-randomness/3705/1. Accessed: 2024-11-12.

[22] Ulysse Pavloff, Yackolley Amoussou-Guenou, and Sara Tucci-Piergiovanni. 2022. Ethereum Proof-of-Stake under Scrutiny. *arXiv preprint arXiv:2210.16070* (2022).

[23] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–214.

[24] Mayank Raikwar and Danilo Gligoroski. 2022. Sok: Decentralized randomness beacon protocols. In *Australasian Conference on Information Security and Privacy*. Springer, 420–446.

[25] Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Pearson.

[26] Caspar Schwarz-Schilling, Joachim Neu, Barnabé Monnot, Aditya Asgaonkar, Ertem Nusret Tas, and David Tse. 2022. Three Attacks on Proof-of-Stake Ethereum. In *Financial Cryptography and Data Security*, Ittay Eyal and Juan Garay (Eds.). Springer International Publishing, Cham, 560–576.

[27] Michael Sproul. 2023. *Allow honest validators to reorg late blocks*. https://github.com/ethereum/consensus-specs/pull/3034

[28] V. Buterin. 2018. RNG exploitability analysis assuming pure RANDAO-based main chain. https://ethresear.ch/t/rng-exploitability-analysis-assuming-pure-randao-based-main-chain/1825 Accessed: 2024-11-12.

[29] Toni Wahrstätter. 2023. Selfish Mixing and RANDAO Manipulation. https://ethresear.ch/t/selfish-mixing-and-randao-manipulation/16081. Accessed: 2024-11-12.

[30] Ben Weintraub, Christof Ferreira Torres, Cristina Nita-Rotaru, and Radu State. 2022. A flash (bot) in the pan: measuring maximal extractable value in private pools. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 458–471.

[31] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[32] Aviv Yaish, Gilad Stern, and Aviv Zohar. 2022. Uncle Maker: (Time)Stamping Out The Competition in Ethereum. Cryptology ePrint Archive, Paper 2022/1020. https://doi.org/10.1145/3576915.3616674

[33] Yaning Zhang and Youcai Qian. 2019. Randao: A DAO working as RNG of Ethereum. https://github.com/randao/randao/. Accessed: 2024-11-12.

A DEFERRED PROOFS

A.1 Proofs for forking attack strings

Theorem (Condition for forking). *Given* $\mathbf{A}^{a_1}$ *slots followed by* $\mathbf{HXA}$, *(where* $\mathbf{X} \in \{\mathbf{A}, \mathbf{H}\}^{h-1}$, $a_1, h > 0$*)* $\mathcal{A}$ *can perform an ex-ante reorg with* $0 < \alpha < 0.5$ *stakes forking out* $\mathbf{HX}$ *if*

$$a_1 \geq \frac{h(1-2\alpha) - p_{\text{boost}}}{\alpha} \quad . \tag{10}$$

Proof. When $\mathcal{A}$ decides to fork the blockchain, the forking attack is successful if the sum of all the votes (sometimes attestations) on $\mathcal{A}$'s blockchain fork is larger than the sum of votes on the honest fork. The adversarial strategy entails the following steps:

(1) $\mathcal{A}$ secretly builds the first adversarial slots $\mathcal{Y}$, and a subset of its remaining $a_1 - 1$ slots. Additionally, all its validators vote at each slot on $\mathcal{Y}$.
(2) Since the honest parties do not see the secret block(s) of $\mathcal{A}$, they build their blocks on top of the parent of block $\mathcal{Y}$ and start a new blockchain fork. Every honest block in $\mathbf{HX}$ is built on this fork, and honest validators vote on them.
(3) $\mathcal{A}$ proposes its block $\mathbf{A}$ on top of the last hidden block following $\mathbf{HX}$, along with its secret fork and corresponding votes.

$\mathcal{A}$'s fork has $a_1\alpha + h\alpha + p_{\text{boost}}$ votes, while the honest parties' blockchain fork accumulated $(1-\alpha)h$ attestations. Honest validators will switch to $\mathcal{A}$'s fork if it has more votes, *i.e.*,

$$a_1\alpha + h\alpha + p_{\text{boost}} \geq (1-\alpha)h \quad . \tag{11}$$

By rearranging Equation (11), and applying that $a_1 > 0$, we obtain the claimed inequality for a_1. □

As a corollary, we define the following quantity:

Definition 10 (Minimum slots to fork). *Let* $\mathsf{minFork}(\alpha, h)$ *represent the minimum value of* a_1 *required for successfully ex-ante reorging* h *blocks given* α *stake. This function combines Theorem 1 and the condition that* $a_1 \in \mathbb{N}^+$.

$$\mathsf{minFork}(\alpha, h) = \max\left(\left\lceil \frac{h(1-2\alpha) - p_{\text{boost}}}{\alpha} \right\rceil, 1\right) \quad . \tag{12}$$

Next we prove Theorem 2: Given any $S \in AS_\alpha$, for a forking string $\mathbf{A}^{a_1}\mathbf{H}^h\mathbf{A}$ where Equation (4) holds, $\mathbf{A}^{a_1}\mathbf{H}^h\mathbf{A}S$ is also an attack string.

Proof. Given $S \in AS_\alpha$ attack string, there is a possible RANDAO value $R^{e,*}$ that can be computed upon reaching the first slot of S. $\mathcal{A}$ can first ex ante reorg $\mathbf{H}^h$ by privately building $\mathbf{A}^{a_1}$, then building on top of it in the last block of the forking string. After $\mathcal{A}$ can follow the same actions required to reach $R^{e,*}$ in S. □

Theorem. *For any forking* eas *of a state* s *and observations* $O \in \Omega$*:*

$$U(s) = \max(\{\mathsf{N}_\alpha(s), \mathsf{F}_\alpha(s, a_1, O), \ldots, \mathsf{F}_\alpha(s, a_n, O)\}) \quad , \tag{13}$$

where $\{a_1, \ldots, a_n\} \subset A(s)$ *are the actions where* $\mathcal{A}$ *starts ex-ante reorging in a specific way.*

Proof. The set of actions $A(s)$ can be divided into two disjoint subsets: actions related to honest behaviour and actions related to ex-ante reorgs. As such,

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') = \max(\{\mathsf{N}_\alpha(s), \mathsf{F}_\alpha(s, a_1, O), \ldots, \mathsf{F}_\alpha(s, a_n, O)\}) \quad . \tag{14}$$

Note that $R(s) = 0$ in these states of the decision tree, as $\mathcal{A}$ is rewarded (and punished for the sacrifice) at the end of epoch e. □

A.2 RANDAO manipulations with ex-ante forking

In this subsection, we define the assumptions under which $\mathcal{A}$ operates in our modeling when conducting an ex-ante forking manipulation. Let the beginning of an attack string as be described by $(\mathsf{a}_1, \mathsf{h}, \mathsf{a}_2) \in \mathbb{N}^{+3}$ such that it starts with $\mathbf{A}^{\mathsf{a}_1}\mathbf{H}^{\mathsf{h}}\mathbf{A}^{\mathsf{a}_2}$ disregarding the epoch boundary. Suppose $\mathcal{A}$ starts ex-ante reorging by first privately building a block allocated in $\mathbf{A}_{\mathsf{a}_1}$ at slot s_p. Note that s_p is not necessarily the first slot of $\mathbf{A}^{\mathsf{a}_1}$. The following simplifying conditions regarding the $\mathcal{A}$'s behaviour hold in our model:

- At slot s_e allocated in $\mathbf{A}^{\mathsf{a}_2}$, $\mathcal{A}$ will finish ex-ante reorging by building upon the latest private block.
- During the slots in $\mathbf{A}^{\mathsf{a}_1}$ after s_p, $\mathcal{A}$ does not publish any blocks, it only misses or privately builds blocks on top of the latest private block.
- In the first slot of $\mathbf{A}^{\mathsf{a}_2}$, s_r, $\mathcal{A}$ might reconsider forking after the last RANDAO reveal of $\mathcal{H}$: r_{s_r-1}. In this case, $\mathcal{A}$ regrets forking and abandons its privately built block(s) and considers $\mathbf{H}^e_{s_r-1}$ as the head of the beacon chain.
- In some extreme cases, $\mathcal{A}$ might prolong the forking if s_e is not the first slot of $\mathbf{A}^{\mathsf{a}_2}$, in which case all slots before s_e are missed. We refer to this phenomenon as *additional sacrifice*.
- $\mathcal{A}$ tries to minimise the number of missed slots during a regretted ex-ante reorg by building the secret fork as late as possible.

Algorithm 1 calculates how $\mathcal{A}$ can fork out a set of honest blocks $\mathbf{H}^h$ in as. The output is a pairs of natural numbers (i, b), where $(0 < b \leq \mathsf{a}_1)$ denotes that $\mathcal{A}$ is able to fork out $\mathbf{H}^h\mathbf{A}^i$ by privately building $\mathbf{A}^b$ blocks preceeding the honest cluster of blocks $\mathbf{H}^h$. Greater additional sacrifices ($\mathbf{A}^i$) may require more blocks (b) to gather enough votes. If $i = 0$, we expect $\mathbf{A}^b\mathbf{H}$ to be in epoch e, furthermore if $i > 0$, then $\mathbf{A}^b\mathbf{H}^h\mathbf{A}^i$ should be in epoch e to manipulate R^e.

Consider the attack string $\mathbf{AHA}^2\cdot$. Then $\mathcal{A}$ with $\alpha = 0.4$ can fork out $\mathbf{H}$ either sacrificing one of its blocks following $\mathbf{H}$ or none of them, *i.e.*, $\textsc{ExAnte}(0.4, 1, 1, 2, 28) = \{(0, 1), (1, 1)\}$.

In Algorithm 3, we calculate the necessary actions during $\mathbf{A}^{\mathsf{a}_1}$ to perform an ex-ante reorg. The input consists of a so-called plan $\in \{\mathbf{C}, \mathbf{N}\}^{\mathsf{a}_1}$, denoting the previsioned statuses of the corresponding $\mathbf{A}^{\mathsf{a}_1}$ slots when forking. $\mathcal{A}$ will minimise the number of privately built blocks as they might be potentially sacrificed during a regretted fork. The minimum number of slots $\mathcal{A}$ has to build privately in $\mathbf{A}^{\mathsf{a}_1}$ is denoted by n, which can be calculated in Algorithm 1. Sometimes the plan is unfeasible, *e.g.*, plan $= \mathbf{N}^{\mathsf{a}_1}$ because clearly, $\mathcal{A}$ is unable to execute an ex-ante reorg without blocks ending up on the canonical chain. In the above case, $\emptyset$ is returned.

Consider the attack string $\mathbf{A}^3\mathbf{HA}\cdot$. Assume that $\alpha \geq 0.2$ and $\mathcal{A}$'s plan $:= \mathbf{C}^2\mathbf{N}$. Then, $\textsc{ForkAction}(\text{plan}, 3, 1) = \textbf{Prop}\cdot\textbf{Hide}\cdot\textbf{Miss}$.

Algorithm 1 Ex-ante reorg evaluation for forking string $\mathbf{A}^{a_1}\mathbf{H}^{h}\mathbf{A}^{a_2}$. Returns a set of tuples (i, b) denoting how many blocks $\mathcal{A}$ must build privately (b) to fork out $\mathbf{H}^{h}\mathbf{A}^{i}$ blocks. If $i > 0$ $\mathcal{A}$ only forks out $\mathbf{H}^{h}\mathbf{A}^{i}$, if it is still in epoch e. Otherwise, we only expect $\mathbf{A}^{a_1}\mathbf{H}$ to be in the current epoch.

procedure ExAnte($\alpha : \mathbb{R}^+, a_1 : \mathbb{N}, h : \mathbb{N}, a_2 : \mathbb{N}, s : \mathbb{N}) \to \mathcal{P}(\mathbb{N} \times \mathbb{N}^+)$
  remain := $32 - s$ ▷ Remaining slots in this epoch
  $C := \{\}$ ▷ $i \in C \Rightarrow i = 0 \vee \mathbf{A}^{a_1}\mathbf{H}^{h}\mathbf{A}^{i}$ is in epoch e
  if $a_1 <$ remain **then**
    $C := \{0\}$; $i := 1$ ▷ $\mathbf{A}^{a_1}\mathbf{H}^{1}$ in epoch e
    while $a_1 + h + i \leq$ remain $\wedge\ i < a_2$ **do**
      $C := C \cup \{i\}$
      $i := i + 1$
    end while
  end if
  ▷ $(i, b) \in C'$: $\mathcal{A}$ can fork out $\mathbf{H}^{h}\mathbf{A}^{i}$ with $\mathbf{A}^{b}$ slots
  $C' := \{(i, \text{minFork}(\alpha, h + i)) \mid i \in C\}$
  return $\{(i, b) \mid (i, b) \in C', b \leq a_1\}$
end procedure

Algorithm 2 Actions described as procedures. For the sake of simplicity, we disregard attestations and assume $\mathcal{A}$ successfully ex-ante reorgs whenever action Fork is called. HEAD denotes the blockchain head considered by the hones validators, while PRIVHEAD is the latest private block built by $\mathcal{A}$ if there is a fork. The current slot number is s. Note, that $\mathcal{A}$ always votes for slot PRIVHEAD.

State Variables: HEAD, PRIVHEAD, s
procedure Propose
  Build block on HEAD and propose it
  if HEAD = PRIVHEAD **then**
    HEAD := PRIVHEAD := s
  else
    HEAD := s
  end if
end procedure
procedure MISS
  pass
end procedure
procedure Hide
  Privately build block on PRIVHEAD
  PRIVHEAD := s
end procedure
procedure Fork
  Build block on PRIVHEAD and propose private blocks
  HEAD := PRIVHEAD := s
end procedure
procedure Regret
  PRIVHEAD := HEAD
end procedure

Algorithm 6 Recursively creating a graph for forking attack strings. Creating an empty graph and populating it with the help of BuildGr.

procedure ForkNodes($\alpha : \mathbb{R}^+$, as : $AS_\alpha(m,n)$, Pr : $\{\mathbf{C},\mathbf{N}\}^*$)
  $V := \{\}; \vec{E} := \{\}$ ▷ Empty graph
  $\text{as}_1 := \text{tail(as)}$; $\text{as}_2 := \text{head(as)}$; $\text{as}' := \text{as}_1 + \text{as}_2$
  Parse $\text{as}' \in \mathbf{A}^{\text{a}_1}\mathbf{H}^{\text{h}}\mathbf{A}^{\text{a}_2}(\mathbf{H}\{\mathbf{A},\mathbf{H}\}^* \cup \{\epsilon\})$
  $\mathcal{E} := \text{ExAnte}(\alpha, \text{a}_1, \text{h}, \text{a}_2, 32 - |\text{as}_1|)$
  $\text{as}_{r,1} := \text{as}_1[\text{a}_1 + \text{h} :]$
  $\text{as}_n := \text{as}_r := \text{as}_{r,1} + \text{“}\cdot\text{”} + \text{as}_2$ ▷ Regret/Not forking
  $\mathcal{R} := \{\}$ ▷ Set of known and relevant statuses
  $C := \{\}$ ▷ Set of vertices with actions to connect later
  for $(i,b) \in \mathcal{E}$ **do** ▷ i sacrifice, b blocks to fork
    $\text{as}_f := \text{as}_1[\text{a}_1 + \text{h} + i + 1 :] + \text{“}\cdot\text{”} + \text{as}_2$
    if $\text{as}_f[0] = \mathbf{H}$ **then**
      continue ▷ RANDAO of $\text{as}_f[0]$ is unknown
    end if
    for plan $\in \{\mathbf{C},\mathbf{N}\}^{\text{a}_1}$ **do**
      act := ForkAction(plan, a_1, b)
      if act = $\emptyset$ **then**
        continue ▷ Not feasible plan
      end if
      map := **Prop**, **Hide**, **Miss** $\rightarrow \mathbf{C},\mathbf{N},\mathbf{N}$
      $r := \text{act.replace(map)}$ ▷ Statuses when regret
      $\text{Pr}_f := \text{plan} + \mathbf{N}^{\text{h}+i} + \mathbf{C}$ ▷ Forking
      $\text{Pr}_r := r + \mathbf{C}^{\text{a}_1}$ ▷ Regret statuses
      $(v_f, V_f, \vec{E}_f) := \text{BuildGr}(\alpha, \text{as}_f, \text{Pr} + \text{Pr}_f)$
      $(v_r, V_r, \vec{E}_r) := \text{BuildGr}(\alpha, \text{as}_r, \text{Pr} + \text{Pr}_r)$
      $\mathcal{R} := \mathcal{R} \cup \mathcal{K}(v_f)$ ▷ $\mathcal{A}$ knows RANDAO of forking
        ▷ Intermediate vertex, where $\mathcal{A}$ started forking:
      $v_i := \text{Vertex}(\text{as}, 32 - |\text{as}_{r,1}|, \mathcal{K}(v_f) \cup \mathcal{K}(v_r))$
      $V := V \cup V_f \cup V_r \cup \{v_i\}$ ▷ Adding vertices
      $e_f := (v_i, v_f)$; $e_r := (v_i, v_r)$
      $\text{Pr}(e_f) := \text{Pr}_f$; $\text{act}(e_f) :=$ **Miss**^{i}**Fork**
      $\text{Pr}(e_r) := \text{Pr}_r$; $\text{act}(e_r) :=$ **Regret**
      $C := C \cup \{(v_i, \text{act})\}$ ▷ Connectivity infos
      $\vec{E} := \vec{E} \cup \vec{E}_f \cup \vec{E}_r \cup \{e_f, e_r\}$
    end for
  end for
  $v_b := \text{Vertex}(\text{as}, 32 - |\text{as}_1|, \mathcal{R})$ ▷ Starting vertex
  for $(v_i, \text{act}) \in C$ **do**
    $e := (v_b, v_i) : \text{Pr}(e) = \epsilon \wedge \text{act}(e) = \text{act}$
    $\vec{E} := \vec{E} \cup \{e\}$ ▷ $\mathcal{A}$ starts forking
  end for
  $(v_n, V_n, \vec{E}_n) := \text{BuildGr}(\alpha, \text{as}_n, \text{Pr} + \mathbf{C}^{\text{a}_1+\text{h}})$
  $e_n := (v_b, v_n)$ ▷ $\mathcal{A}$ does not fork
  $\text{Pr}(e_n) := \mathbf{C}^{\text{a}_1+\text{h}}$; $\text{act}(e_n) :=$ **Prop**$^{\text{a}_1}$
  $V := V \cup V_n \cup \{v_b\}$; $\vec{E} := \vec{E} \cup \vec{E}_n \cup \{e_n\}$
  return $(v_b, V, \vec{E})$
end procedure

Algorithm 4 Constructing a graph from as attack string, using the helper functions: Algorithms 5 and 6. Whenever SelfishMixingNodes returns an empty string, we know that $\mathcal{A}$ might be able to fork, thus we call ForkNodes.

procedure BuildGr($\alpha : \mathbb{R}^+$, as : $AS_\alpha(m,n)$, Pr : $\{\mathbf{C},\mathbf{N}\}^*$)
  $(v, V, \vec{E}) :=$ SelfishMixingNodes(α, as, Pr)
  if $V = \emptyset$ **then**
    $(v, V, \vec{E}) :=$ ForkNodes(α, as, Pr)
  end if
  return $(v, V, \vec{E})$
end procedure

Algorithm 5 Constructing a graph from as expecting it to be a simple selfish mixing or “empty” string, where no manipulation is possible (*e.g.*, as $\in$ “ $\cdot$ ” + $\{\mathbf{A},\mathbf{H}\}^n$). Otherwise, an empty graph is returned.

procedure SelfishMixingNodes($\alpha : \mathbb{R}^+$, as : $AS_\alpha(m,n)$, Pr : $\{\mathbf{C},\mathbf{N}\}^*$)
  $v_b := nil$; $V := \{\}$; $\vec{E} := \{\}$
  Parse as_1, as_2 such that $\text{as} = \text{as}_1 + \cdot + \text{as}_2$
  if $\text{as}_1 = \epsilon$ **then** ▷ Empty string
    $v_b := \text{Vertex}(\text{as}, 32, \{\text{Pr}\})$
    $V := \{v_b\}$ ▷ Single vertex, $\mathcal{A}$ cannot attack
  else if $\exists t \in \mathbb{N}^+ : \text{as}_1 = \mathbf{A}^t$ **then** ▷ Selfish mixing
    $(v_m, V_m, \vec{E}_m) := \text{BuildGr}(\alpha, \text{as}[1:], \text{Pr} + \mathbf{N})$
    $(v_p, V_p, \vec{E}_p) := \text{BuildGr}(\alpha, \text{as}[1:], \text{Pr} + \mathbf{C})$
    $v_b := \text{Vertex}(\text{as}, 32 - |\text{as}_1|, \mathcal{K}(v_m) \cup \mathcal{K}(v_p))$
    $e_m := (v_b, v_m)$; $e_p := (v_b, v_p)$ ▷ Edges
    $\text{Pr}(e_m) := \mathbf{N}$; $\text{Pr}(e_p) := \mathbf{C}$ ▷ Printer
    $\text{act}(e_m) :=$ **Miss**; $\text{act}(e_p) :=$ **Prop** ▷ Actions
    $V := V_m \cup V_p \cup \{v_b\}$; $\vec{E} := \vec{E}_m \cup \vec{E}_p \cup \{e_m, e_p\}$
  end if
  return $(v_b, V, \vec{E})$
end procedure

A.3 Decision trees

In this subsection, we show how one can construct a decision tree $G = (V, \vec{E})$ based on an attack string as. A state was previously described with the help of extended attack strings, but for the sake constructing decision trees, the postfix of epoch $e+1$ is irrelevant. Although Algorithm 4 handles the overall tree construction, the concrete details of the graph structure are developed within two helper functions: Algorithms 5 and 6. All 3 of these algorithms return a graph in the following format: $(v, V, \vec{E})$, where $v \in V$ is the root of the constructed tree $(V, \vec{E})$. We define the following functions on G, assigned during the construction of the decision tree:

- as : $V \rightarrow AS_\alpha(m,n)$,
- $\mathcal{K} : V \rightarrow \mathcal{P}(\{\mathbf{C},\mathbf{N}\}^m)$,
- s : $V \rightarrow \mathbb{N}$,
- Pr : $\vec{E} \rightarrow \{\mathbf{C},\mathbf{N}\}^*$,
- act : $\vec{E} \rightarrow A$.

Each state ($v \in V$) is described by an attack string as(v), the current slot number s(v) and finally the set of realisation strings, where the corresponding $R^{e,*}$ can be computed. $\mathcal{A}$ can then choose an action act(e) according to the optimal policy *cf.* Section 4.2.3.

Once $\mathcal{A}$ executes a **Fork** or **Regret** action, the rest of the attack can be described by the remaining slot statuses of epoch e. The canonical chain is agreed on by $\mathcal{H}$ and $\mathcal{A}$ and $\mathcal{A}$ does not try to fork out these blocks, rather focusing for the rest of the epoch. We account for these agreed blocks with the function called printer ($\Pr(e)$) outputting slot statuses (C/N) corresponding to the initial characters of $\text{as}(v)$. The term printer reflects the irreversible nature of these outputs, as each status is fixed once produced. For the sake of concreteness, suppose v is the state described by $\mathbf{AHA}\cdot$, after $\mathcal{A}$ started forking, and needs to decide whether to finish forking, or building on top of $\mathbf{H}_{30}$. Let $e = (v, v')$ correspond to the decision of regretting forking, the above discussed functions are yielding the following values:

- $\text{as}(v) = \mathbf{AHA}\cdot$
- $\mathcal{K}(v) = \{\mathbf{CNC}, \mathbf{NCC}, \mathbf{NCN}\}$
- $\text{s}(v) = 31$
- $\Pr(e) = \mathbf{NC}$
- $\text{act}(e) = \textbf{Regret}$

Algorithm 3 Calculating the actions needed to perform an ex-ante reorg given a plan : $\{\mathbf{C}, \mathbf{N}\}^{\text{a}_1}$, which denotes the desired statuses of the first $\mathcal{A}$ cluster. Additionally, $0 < \text{n} \leq \text{a}_1$ means the number of necessary slots needed to ex-ante reorg (see Algorithm 1).

procedure ForkAction(plan : $\{\mathbf{C}, \mathbf{N}\}^{\text{a}_1}, \text{a}_1 : \mathbb{N}, \text{n} : \mathbb{N}) \rightarrow \mathcal{B}^{\text{a}_1} \cup \{\emptyset\}$
  $C := \{i \in \mathbb{N} \mid i \leq \text{a}_1 - \text{n} \wedge \text{plan}[i] = \mathbf{C}\}$ ▷ Candidate indices
  if $C = \emptyset$ **then** ▷ $i \leq \text{a}_1 - \text{n} : \text{plan}[i] = \mathbf{N} \rightarrow$ no voting here
    return $\emptyset$ ▷ Votes in the remaining $\text{n} - 1$ slots $\emptyset$ enough
  end if
  $l := \max(C)$ ▷ $\mathcal{A}$ starts forking as late as feasible
  $\text{plan}_1 := \text{plan}[: l]$ ▷ $\mathcal{A}$ proposes blocks during the C slots
  $\text{plan}_2 := \text{plan}[l :]$ ▷ $\mathcal{A}$ privately builds during the C slots
  $\text{act}_1 := \text{plan}_1.\text{replace}(\mathbf{C}, \mathbf{N} \rightarrow \textbf{Prop}, \textbf{Miss})$
  $\text{act}_2 := \text{plan}_2.\text{replace}(\mathbf{C}, \mathbf{N} \rightarrow \textbf{Hide}, \textbf{Miss})$
  return $\text{act}_1 + \text{act}_2$
end procedure

For the sake of simplicity, let $\text{Vertex}(\text{as}, s, \mathcal{R})$ denote the operation of creating a vertex with the following attributes: $\text{as}(v) = \text{as} \wedge \text{s}(v) = s \wedge \mathcal{K}(v) = \mathcal{R}$.

Algorithm 5 takes α stake, as attack string and the already printed characters Pr. This function is responsible only for managing simple cases where no manipulation or selfish mixing is possible.

- **Empty string:** $\text{tail}(\text{as}) = \epsilon$. $\mathcal{A}$ cannot manipulate in either case due to the absence of A slots in epoch e.
- **Selfish mixing:** $t \in \mathbb{N}^+ : \text{tail}(\text{as}) = \mathbf{A}^t$. $\mathcal{A}$ either **Miss**/**Prop**, the algorithm recursively calls back to the $\mathbf{A}^{t-1}$+"·"+head(as) attack string, which is either *selfish mixing* or an empty attack string.
- **Other:** Empty graph is returned. BuildGr then calls Algorithm 6 as it is a forking string.

In Algorithm 6, we construct forking decision trees by using recursion. The first step is extracting the values $(\text{a}_1, \text{h}, \text{a}_2)$ such that as starts with $\mathbf{A}^{\text{a}_1}\mathbf{H}^{\text{h}}\mathbf{A}^{\text{a}_2}$. $\mathcal{A}$ can either:

- **Do not fork:** In this scenario $\mathcal{A}$ executes the actions $\textbf{Prop}^{\text{a}_1}$. The attack string as_n describes the remaining slots, thus BuildGr is called to construct this branch recursively. Characters $\mathbf{C}^{\text{a}_1+\text{h}}$ are printed. We show that if $\mathcal{A}$ starts forking, there will be at least one N slot in the interval $\mathbf{A}^{\text{a}_1}\mathbf{H}^{\text{h}}$, thus these branches of the tree are independent.
- **Starts forking:** $\mathcal{A}$ evaluates which attacks it can pursue with Algorithm 1. The algorithm iterates through the possible values of $(\text{plan}, i) \in \{\mathbf{C}, \mathbf{N}\}^{\text{a}_1} \times \mathbb{N}$, where plan denotes the intended statuses of $\mathbf{A}^{\text{a}_1}$, while $\mathbf{H}^{\text{h}}\mathbf{A}^i$ the interval which $\mathcal{A}$ plans to fork out. Algorithm 3 yields the action (act) leading to a state after $\mathbf{H}^{\text{h}}$ (v_i), where $\mathcal{A}$ has to choose between:
  - Finishing the described attack by missing i blocks, then building on top of the latest private block (**Miss**^{i}**Fork**). Because the plan was achieved, the slot statuses plan + $\mathbf{N}^{h+i}\mathbf{C}$ are added to the finalised statuses (Pr). The attack string as_f describes the state of advancing $\text{a}_1 + \text{h} + i + 1$ slots. The rest of the tree is constructed recursively from as_f.
  - Regretting the attack by abandoning the privately built blocks, which results at least one missed slot in $\mathbf{A}^{\text{a}_1}$. Hidden and missed blocks contribute to Pr as N, while proposed blocks as C. The attack continues recursively at v_r, described by the attack string as_r, after $\mathbf{C}^{\text{h}}$ is concatenated at the end of Pr. This branch is independent of the above branch, as $\mathbf{H}^{\text{h}}$ will be part of the canonical chain here.

B ADDITIONAL OBJECTIVE FUNCTIONS FOR RANDAO MANIPULATIONS

As we alluded to in Section 2.3, a RANDAO manipulator might be motivated by other objective functions than just maximizing the number of obtained slots. In particular, one objective function is the *target slot utility*, where the adversary is motivated to propose a specific slot in the next epoch. More formally, we define the target-slot reward function Γ_π^{tslot} for validator i and the j^{th} slot in epoch $e + 2$ as:

$$\Gamma_\pi^{tslot} := I(\mathbf{v}_R^{e+2}[j] == i) \ , \tag{15}$$

where $I(\cdot)$ is an indicator variable.

As an extension, one might also consider a forward-looking objective where the validator aims only to maximize the number of blocks obtained in epoch $e + 2$ discounting any losses incurred by the manipulation strategy in epoch e, that is

$$\max_{R^e} p(e + 2, R^e, \mathcal{A}) \ . \tag{16}$$

In addition, one might consider various other manipulation objectives. However, this work focuses on the utility functions defined in Equations (3) and (15), disregarding any other incentives validators may have. We leave the exploration of different utility functions to future work.

B.1 Evaluating the target slot utility

We evaluated the manipulative power of an adversary that wishes to optimize for the target slot utility function, *cf.* Equation (15). In particular, we consider a model in which the adversary manipulates the RANDAO *once* in epoch e to obtain a specific slot, say the

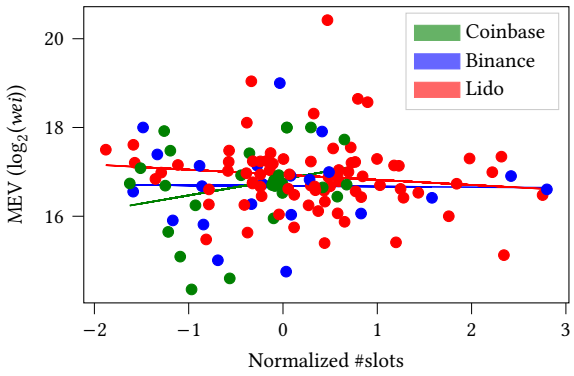


Figure 13: MEV incentives of three major validators whenever they ex-post reorged blocks at the end of an epoch. We normalised the slots obtained in the next but one epoch by the entities' stake and the MEV of the reorging block. We see no correlation between the MEV content of the reorged blocks and the normalised RANDAO outcome.

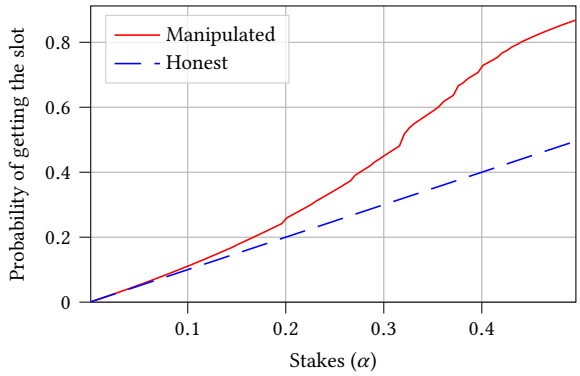


Figure 11: The expected value of obtaining a target slot for an honest player (blue) and a RANDAO manipulator using selfish mixing and forking strategies.

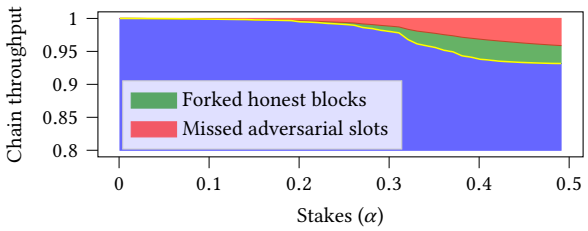


Figure 12: Blockchain throughput degradation as a function of the RANDAO manipulator adversary's staking power α.

i^{th} slot in epoch $e+2$. We evaluate the adversarial probability of this "one-shot" RANDAO manipulation to obtain the desired slot in Figure 11. We find that, for instance, for $\alpha = 0.3$, the adversary has a 0.451 probability of obtaining a desired target slot in epoch $e+2$. We leave it to future work to study the target slot utility function in other, perhaps stronger adversarial models. For example, one could study the target slot utility in an adversarial model in which $\mathcal{A}$ manipulates the RANDAO multiple epochs before the desired target epoch to enhance its manipulative power and increase its probability of obtaining the target slot.

C ADDITIONAL EMPIRICAL MEASUREMENTS

In this section, we provide measurements that we could not include in the main text due to space constraints.

C.1 Measurements about the forking policy

First, we show in Figure 12, how a forking RANDAO manipulator staking entity decreases the blockchain's throughput by forking numerous honest blocks out.

C.2 Forking and MEV: are they related?

We collected ex-ante and ex-post reorgs executed by three major entities (*i.e.*, Lido, Coinbase, Binance) where they reorged a tail slot $\mathbf{H}_{30}$ or $\mathbf{H}_{31}$. For each entity and tail slot reorg event $\mathbf{HA}\cdot$ or $\mathbf{H}\cdot\mathbf{A}$, we created a pair $(p(e+2, R^e, \mathcal{A}), f)$, where f is the amount the block builder paid to the validator in the block $\mathbf{A}$. Since it is hard to assess the MEV content of a block, we use f as a proxy to approximate the MEV content of a block. We do not observe a significant correlation between the MEV content of the block $\mathbf{A}$ and the number of slots obtained by the ex-post forking entity in epoch $e+2$, see Figure 13. The observed correlation coefficients are $-0.13, -0.02, 0.26$ for Lido, Binance, and Coinbase, respectively. This indicates that, as of the time of writing, ex-post forking decisions show no significant correlation with RANDAO manipulation considerations.