BunnyFinder: Finding Incentive Flaws for Ethereum Consensus

Rujia Li*¶, Mingfei Zhang†, Xueqian Lu, Wenbo Xu‡, Ying Yan‡, Sisi Duan*§¶ ✉
*Tsinghua University †Shandong University ‡Blockchain Platform Division, Ant Group
§Zhongguancun Laboratory, Shandong Institute of Blockchains
¶State Key Laboratory of Cryptography and Digital Economy Security
rujia@tsinghua.edu.cn, mingfei.zh@outlook.com, xueqian.lu@bitheart.org,
{xuwenbo.xwb,fuying.yy}@antgroup.com, duansisi@tsinghua.edu.cn
✉ Corresponding author

***Abstract*—Ethereum, a leading blockchain platform, relies on incentive mechanisms to improve its stability. Recently, several attacks targeting the incentive mechanisms have been proposed. Examples include the so-called reorganization attacks that cause blocks proposed by honest validators to be discarded to gain more rewards. Finding these attacks, however, heavily relies on expert knowledge and may involve substantial manual effort.**

We present BunnyFinder, a semi-automated framework for finding incentive flaws in Ethereum. BunnyFinder is inspired by failure injection, a technique commonly used in software testing for finding implementation vulnerabilities. Instead of finding implementation vulnerabilities, we aim to find design flaws. Our main technical contributions involve a carefully designed "strategy generator" that generates a large pool of attack instances, an automatic workflow that launches attacks and analyzes the results, and a workflow that integrates reinforcement learning to fine-tune the attack parameters and identify the most profitable attacks. We simulate a total of 9,354 attack instances using our framework and find the following results. First, our framework *reproduces* five known incentive attacks that were previously found manually. Second, we find three new attacks that can be identified as incentive flaws. Finally and surprisingly, one of our experiments also identified two implementation flaws.

I. Introduction

Ethereum is a leading blockchain platform [1], [2] with a market cap of 19 billion dollars [3]. After it upgraded to the Proof-of-Stake (PoS) consensus mechanism in Sep 2022, it now heavily relies on the *incentive mechanisms* to ensure the stability of the system. Namely, honest validators receive rewards, and Byzantine validators (i.e., arbitrarily faulty validators) may receive penalties, so validators are encouraged to follow the specifications of the protocol. The rewards include the *block rewards*, *transaction fees*, and *attestation rewards* (i.e., every honest validator that *votes* correctly receives rewards), etc. The penalties include the *slashing conditions* (i.e., a validator that equivocates will be penalized and eventually removed from the system) and *attestation penalties* (i.e., a validator that does not vote will suffer from penalties). The incentive mechanism has been very successful since the launch of the PoS protocol. According to *rated.network*[1], more than 99% of validators operate correctly.

The relationships between the incentive mechanisms and security of the system. The incentive mechanism of Ethereum is not perfect. Many attacks have been proposed on finding the incentive *flaws* of Ethereum PoS [4]–[7] (see Table I), some of which might pose a security threat to the system. For instance, the recently proposed staircase attack [6] has demonstrated that by controlling more than 29.6% stake, all honest validators may suffer from no attestation incentives, but Byzantine validators continue to receive rewards. Furthermore, the attack can be launched continuously. Eventually, the fraction of stake controlled by the adversary exceeds the pre-specified 33.3% threshold, posing a security threat to the *safety* (i.e., no double spending) and *liveness* (i.e., transactions are eventually finalized) properties of the system [8]. In fact, even if the attack on the incentive mechanism does not have an impact on the security of the system, an *inappropriate* incentive mechanism may discourage honest validators from participating in the system [9], lowering the robustness and reliability of the system.

Finding the incentive flaws is challenging. It is not easy to find incentive flaws (e.g., honest validators receive much lower rewards than the fair share), and existing works largely rely on expert knowledge and may involve substantial manual effort. This difficulty mainly stems from two factors. First, while formal models of blockchain incentives have been studied [13], [14], to our knowledge, no consensus mechanism has incorporated incentives in their security definitions. It is thus extremely challenging, if not possible, to rule out incentive flaws. Second, the attack strategies can be very sophisticated. The staircase attack [6] mentioned above is a perfect example. In the attack, Byzantine validators collude and carefully control the concrete *timing* when the withheld blocks and attestations can be released. As a Byzantine validator may behave arbitrarily, enumerating *all* malicious behaviors to discover new attacks is technically infeasible.

[1]Rated.work: https://explorer.rated.network/ (accessed in Nov 2024)

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230281
www.ndss-symposium.org

TABLE I: Comparison of known incentive attacks and newly found incentive attacks by BunnyFinder.

Scheme	Flaw Type	Attack Strategy		Attack Result		Identified Ethereum Implementations
		Content Manipulation	Order Manipulation	Honest	Byzantine	
Ex-ante reorg attack [5]	I	block	head (short for h)	less reward	reward	Prysm 5.2.0; Teku 25.6.0
Sandwich reorg attack [10]	I	block	head, modify parent	less reward	reward	Prysm 5.2.0; Teku 25.6.0
Unrealized justification attack [11]	III	-	parent	penalty	reward	Prysm 4.0.5
Justification withholding attack [12]	III	block	head	penalty	reward	Prysm 4.0.5
Staircase attack [6]	III	block	source, h target, attestations	penalty	reward	Prysm 4.0.5
Selfish mining attack (this work)	I	block	head, modify parent	less reward	reward	Prysm 4.0.5
Staircase attack-II (this work)	III	block	source, h, target, attestations	penalty	reward	Prysm 5.2.0
Pyrrhic victory attack (this work)	II, IV	blocks, attestations	all	less reward&penalty	penalty	Prysm 5.2.0; Teku 25.6.0

Thus, an interesting and meaningful research question is:

Can we find the incentive flaws in Ethereum while involving less manual effort?

Our Approach. We propose BunnyFinder, a semi-automated framework to find incentive flaws for the Ethereum PoS protocol. BunnyFinder is inspired by *software testing* tools such as penetration testing [15]–[18], chaos engineering [19], [20], fuzz testing [21]–[24], and breach and attack (BAS) simulation [25]. Different from software testing tools that often focus on implementation vulnerabilities or bugs, our framework focuses on the design flaws, especially incentive flaws. As summarized in Figure 1, our framework consists of four components: *strategy generator*, *strategy executor*, *state analyzer*, and *strategy optimizer*. We use the *strategy generator* to automatically generate attack instances. The *strategy executor* automatically executes the corresponding attack strategies. The *state analyzer* collects execution logs to determine whether an incentive flaw has been triggered.

Based on the identified incentive flaws, we further build a *strategy optimizer*, using reinforcement learning [26] to fine-tune the attack parameters, e.g., how much time each message should be delayed. In this way, we can further refine the attack strategies to either increase the probability of launching the attacks or increase the "profit" of the attacks.

Responsible disclosure. We have disclosed our findings to the Ethereum Foundation via both public channels, meetings, and emails. All our discovered vulnerabilities have been reported and acknowledged. See Appendix A for more information.

Our findings. We implement our framework on two widely adopted Ethereum implementations, Prysm [27] and Teku [28]. Using our framework, we generate and run 9,354 attack instances, and all experiments last for about 120 hours. While many attack instances can be defined as incentive flaws, we identify ten *useful* attacks, including five known reorganization attacks, three *new* attacks (as design flaws), and two implementation flaws in Prysm, as summarized below. To further clarify, we use *incentive flaws* to denote design issues, and *implementation flaws* to denote implementation issues that are not directly related to the design. See Table I for a summary.

▷ *Finding 1.* Our framework can *cover* most known reorganization attacks. Since some of the known attacks (e.g., the sandwich reorg attack [10], [29], the unrealized justification attack [11], the justification withholding attack [12], and staircase attack [6]) have already been mitigated in the recent *Deneb* upgrade of the codebase, we can reproduce these attacks on older versions.

▷ *Finding 2.* Our framework identifies three new attacks of the incentive flaws, as summarized below.

(1) We find a *selfish mining* attack, previously known as an attack on Proof-of-Work (PoW) and some PoS protocols [30]–[33] but not Ethereum PoS [31]. To our knowledge, this is the first known selfish mining attack to Ethereum PoS. After manual investigation, we show that by controlling only 13.4% stake, the attack is already profitable, i.e., Byzantine validators gain more rewards than their fair share.

(2) We find a variant of staircase attack, and we call it *staircase attack-II*. In this attack, honest validators suffer from penalties while Byzantine validators still continue to receive rewards. We show that by controlling 33.3% stake, the attack can be launched so all honest validators suffer from penalties. Different from the staircase attack that can be launched continuously in every epoch, staircase attack-II can only be launched for one more epoch with $1/9$ probability. Although Ethereum has provided a mitigation to the staircase attack, our result shows that the solution cannot fully mitigate its variant.

(3) We find a novel attack called *pyrrhic victory attack*[2]. In our attacks, honest validators receive lower rewards than their fair share or even suffer from penalties, while Byzantine validators also suffer from penalties [34]. We found many meaningful attack instances where honest validators suffer from higher incentive loss than Byzantine validators.

▷ *Finding 3.* Although BunnyFinder is not intended to identify implementation vulnerabilities, our framework uncovers two such issues. First, we observe a deadlock in the *synchronization* module, which validators use to "catch up" with other validators, but no validator is able to complete synchronization. The second issue happens in the *log* module, where non-existent reorganization is reported in the log. While this does not affect the correctness of the system, it impacts our state analyzer and may complicate maintenance for developers.

Contributions. Our work makes the following contributions:

[2]A pyrrhic victory means a win that comes at such a great cost to the victor that it is nearly equivalent to defeat.

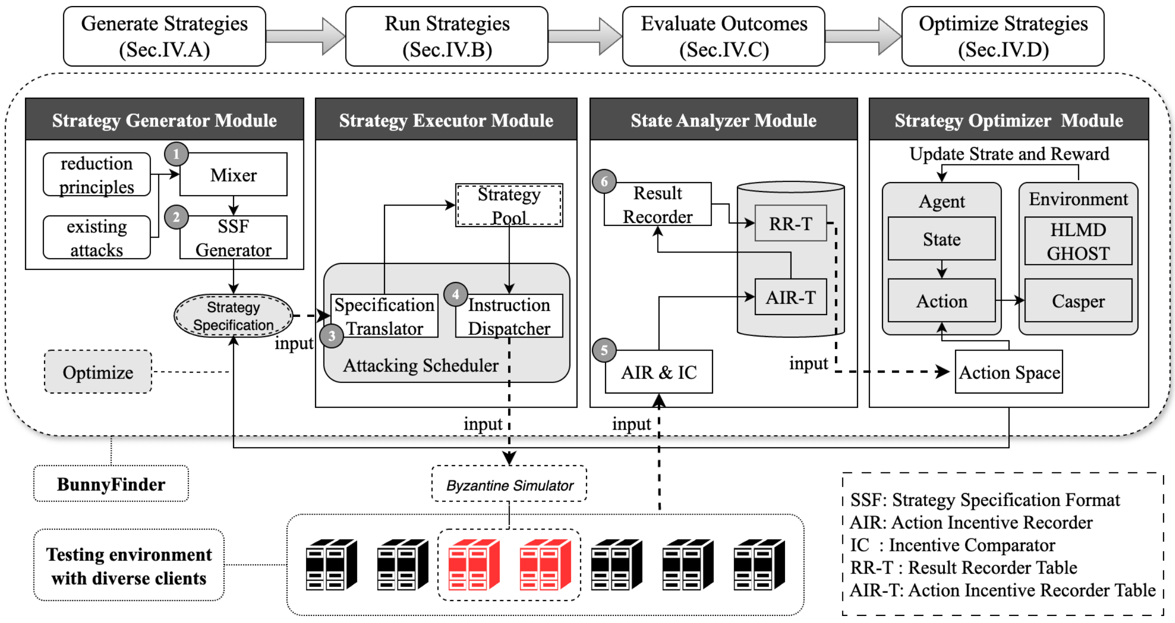

Generate Strategies (Sec.IV.A)
Run Strategies (Sec.IV.B)
Evaluate Outcomes (Sec.IV.C)
Optimize Strategies (Sec.IV.D)
Strategy Generator Module
reduction principles
Mixer
existing attacks
SSF Generator
Strategy Specification
Optimize
Strategy Executor Module
Strategy Pool
Specification Translator
Instruction Dispatcher
Attacking Scheduler
input
State Analyzer Module
Result Recorder
RR-T
AIR-T
AIR & IC
input
Strategy Optimizer Module
Update Strate and Reward
Agent
State
Action
Environment
HLMD GHOST
Casper
Action Space
BunnyFinder
Byzantine Simulator
Testing environment with diverse clients
SSF: Strategy Specification Format
AIR: Action Incentive Recorder
IC : Incentive Comparator
RR-T : Result Recorder Table
AIR-T: Action Incentive Recorder Table

Figure 1: Overview of our work.

- We propose BunnyFinder, a framework for finding the incentive flaws of the Ethereum PoS protocol. BunnyFinder exploits the idea of failure injection to build a workflow that automatically imports pre-defined attack instances, launches an experiment that simulates the attack, and analyzes whether an incentive flaw exists.
- We implement and evaluate our framework on two widely adopted Ethereum implementations. By running the 9,354 attack instances, our framework automatically covers five known reorganization attacks (although only one attack is still *feasible* in the current version of Ethereum) and three new attacks on the incentives of Ethereum. Additionally, as a by-product, our framework identifies two implementation flaws.
- We conduct an in-depth analysis of the newly discovered attacks. All new attacks we found have been validated and reported. Some of them have been acknowledged by the Ethereum Foundation and contributed to version updates in the Ethereum protocol.

II. Ethereum PoS and its Incentive

In this section, we briefly review the Ethereum PoS protocol. Our notations mainly follow the Ethereum documentation [2], [35] and previous works [6].

Epoch and slot. Ethereum PoS assumes that the network is partially synchronous, i.e., there exists an unknown upper bound Δ for message propagation and processing. In the paper, our analysis focuses on the attacks when the network is synchronous, assuming that Δ is known by all validators. In Ethereum, time is divided into epochs. Each epoch includes 32 *slots* where each slot lasts for 12 seconds. For our purpose, we call four seconds (1/3 of a slot) a time *unit*.

Validators. Validators are the parties that participate in the PoS protocol. Without loss of generality, we assume that there are n validators $\{v_1, v_2, ..., v_n\}$ while in practice n may change over time. Each validator deposits 32 ETH as its stake. The stake of each validator represents its *weight*, which is useful during voting. For simplicity, we assume the weights of all validators are the same.

Among n validators, up to f become Byzantine and fail arbitrarily. Since we assume the weights of all validators are the same, and the number of validators is fixed, Ethereum assumes $n \geq 3f+1$, which is optimal for partially synchronous consensus protocols.

In this work, we consider an adversary that may control up to f validators, but the actual number of validators controlled might be lower than f.

Block and Attestation. A block b consists of the slot number, a hash pointer to the *parent* block in the chain, a batch of transactions, and a set of *attestations*. Attestations are the votes of the validators. Each attestation att includes the slot number, the hash of some block, and two hashes of the *checkpoint* (cp) blocks, where a checkpoint is a specific block proposed at a certain point of the protocol.

Proposer and attestor. In each epoch, validators are divided into 32 committees and each committee is assigned to only one slot. In each slot, all validators in the corresponding committee become *attestors* and become eligible to vote. Also, a single validator is chosen to be the *proposer* that is eligible to propose a block. The identities of the committees and proposers are randomly selected and all validators can verify the identities of each other.

Finality. Ethereum PoS uses Casper FFG (Friendly Finality Gadget) [35] to *finalize* the blocks. A block is considered finalized when it has received a sufficient number of attestations.

```
The Ethereum Protocol for Validator $v_i$
global parameters: slot $t$
local parameters: block tree $\mathcal{T}$, attestation set $Atts$.
-----------------------------------------------------------------
01 upon the beginning of a slot $t$ do
02     as the proposer for slot $t$
03     let $head$ be the leaf block of the canonical chain
04     let $atts$ be the set of newly received attestations
05     let $txs$ be the set of newly received transactions
06     build a block $b = (\text{PROPOSE}, t, v_i, H(head), atts, txs)$
07     Broadcast $b$ to all validators
-----------------------------------------------------------------
08 upon four seconds of a slot $t$ do
09     as the attestor for slot $t$
10     let $head$ be leaf block of the canonical chain
11     let $cp_1$ be the first checkpoint
12     let $cp_2$ be the last checkpoint
13     create $att = (\text{ATTEST}, t, v_i, H(head), H(cp_1), H(cp_2))$
14     Broadcast the attestation $att$ to all validators
-----------------------------------------------------------------
15 upon receiving a block $b$ from the proposer $v_j$ do
16     $\mathcal{T} \leftarrow \mathcal{T} \cup b$
-----------------------------------------------------------------
17 upon receiving $att$ from validator $v_j$ do
18     $Atts \leftarrow Atts \cup \{att\}$
```

Figure 2: The Ethereum PoS Protocol. Codes are shown for validator v_i. $H()$ denotes the hash function.

Once a block is finalized, all blocks led by the block will never be reversed.

Fork choice. Ethereum PoS uses the HLMD GHOST (Hybrid Latest Message Driven Greedy Heaviest Observed Sub-Tree) fork choice rule [36] to determine the *canonical chain*. Informally, every proposer extends its canonical chain when it proposes a new block, and every attestor votes for its canonical chain. The HLMD GHOST rule selects the *heaviest* chain based on the number of attestations. We omit the details as it is not relevant to our framework.

Workflow. We show the pseudocode in Figure 2. The protocol proceeds in epochs and each epoch has 32 slots. At the beginning of a slot t, a proposer v_i creates a block $b = (\text{PROPOSE}, t, v_i, H(head), atts, txs)$, where $head$ is the output of the fork choice, $atts$ is a set of attestations, and txs is the batch of transactions (line 1-7). After four seconds of slot t has elapsed, each eligible attestor v_i creates an attestation $att = (\text{ATTEST}, t, v_i, H(head), H(cp_1), H(cp_2))$ and sends att to all validators in the same committee. Here, $head$ is the output of the fork choice, and cp_1 and cp_2 are two checkpoint blocks (line 8-14). After receiving a message m from other validators, m is added into each validator v_i's local block tree $\mathcal{T}$ (line 15-18), which is useful only for the fork choice rule.

The incentive mechanism. The incentive mechanism of Ethereum includes both rewards and penalties. Rewards include block rewards, transaction fees, and attestation fees. Block rewards are rewards each block proposer can receive after its proposed blocks are finalized. Transaction fees are the extra fees users pay to complete a transaction [37]. Attestation fees are the fees each attestor receives after its attestations are finalized. Notably, the concrete values of the attestation rewards are related to the *participation rate*, i.e., informally speaking, the total stake that votes for the same block. For example, if attestors that own 75% stake vote for the same block and their attestations are finalized, these attestors receive 75% of the maximum attestation reward. Penalties, on the other hand, include the slashing condition and attestation penalties. The slashing condition penalizes validators that equivocate (i.e., validators that propose two conflicting blocks or vote for two conflicting branches). Attestation penalties penalize attestors whose attestations are not finalized (while they are supposed to send these attestations).

We call both attestation rewards and attestation penalties *attestation incentives*. Our current work focuses on the attestation incentives and block rewards.

III. Overview of BunnyFinder

A. BunnyFinder in a Nutshell

As mentioned in the introduction and also shown in Figure 1, our framework consists of four main components: the *Strategy Generator* (SG), *Strategy Executor* (SE), *State Analyzer* (SA), and *Strategy Optimizer* (SO). Briefly speaking, the framework proceeds in two major phases.

Phase 1: Attack identification. In the first phase, the SG generates a set of attack instances; the SE imports these attack instances into our locally deployed Ethereum testnet where a fraction of validators apply the corresponding attack strategies; the SA collects the logs from the validators and analyzes whether each attack instance reveals an incentive flaw. These three components are automatically conducted.

Phase 2: Attack optimization. After some incentive flaws are identified, we manually investigate the execution logs and classify the flaws into either existing attacks or new attack types. For those new attack types, we extract the common attack strategies (e.g., withholding a block for a certain number of slots), and then run SO to optimize the attack strategies so that the attack is either easier to launch or more profitable. To optimize the strategies, we use reinforcement learning (RL), a machine learning approach that is unique for performing optimization without training data. Attack optimization requires expert knowledge and is not fully automatic.

B. The Challenges and our Solutions

The main research challenge we address is finding effective attack strategies from a large space of attack instances and then fine-tuning the attack parameters. Briefly speaking, there are two types of malicious behaviors of Byzantine validators: *order manipulation* and *content manipulation*. Order manipulation changes the order of the messages each validator sends to other validators (by Byzantine validators). Meanwhile, content manipulation changes the content of the messages.

Although there are only two types of malicious behaviors, the space of the concrete attack instances is enormous. Indeed, each message may be modified arbitrarily and delayed for an arbitrarily long time. Additionally, the adversary may control different fractions of Byzantine validators. As the fraction of validators controlled by the adversary grows, the size of the message queues that need to be manipulated also grows. As we cannot exhaustively enumerate every attack strategy, we need to significantly reduce the space of the attack instances. Furthermore, even if an attack can be identified as an incentive flaw, we still cannot determine whether such an attack is the most *profitable* one. Indeed, by slightly modifying the attack strategies, the attack may cause higher impact.

Thus, we carefully design three components when designing our BunnyFinder framework: strategy generator (SG), state analyzer (SA), and strategy optimizer (SO).

SG: Reducing the space of attack instances. To reduce the space of attack instances, we introduce several principles for the SG to generate *effective* attack strategies.

- (Not releasing the message earlier than expected.) In the implementation of Ethereum, validators do not process the messages with a slot number higher than its current local slot number. Therefore, releasing a message earlier than expected is meaningless.
- (Setting up discrete delay time.) We set up the delay time of any message discretely where *one unit* is one-third of each slot, i.e., four seconds. We choose this optimization for two reasons. First, every validator triggers some functions at each *unit*, e.g., line 08 of Figure 2 shows that each attestor prepares its attestation after four seconds have elapsed in each slot. Second, we consider a synchronous network and all messages are delivered *on time*. This is in fact a weaker assumption than other timing models. Namely, if an attack can be launched even when the network is synchronous, it can easily be launched when the network is asynchronous.
- (Carefully selecting fields for content manipulation.) We carefully select the fields that can be modified in both the block and the attestation data structures, such that the attack strategies might be effective.

By adopting the principles above, we can greatly reduce the number of concrete attack instances. Jumping ahead a little bit, our SG generates 9,354 attack instances (out of 128^{96} theoretical attack instances), among which 3,121 are identified as incentive flaws according to our criteria.

Our SG approach is heuristic, so we can drastically reduce the size of attack instances from a large space. We acknowledge that novel attacks might be *filtered*. Unfortunately, we are not aware of any approach that can find novel attacks easily. We leave it an interesting future work.

SA: Evaluating the attack instances. Before we identify an incentive flaw, we need to formally define an *incentive flaw*. According to previous work [38], [39], a good incentive mechanism, in general, should satisfy the following conditions:

(1) Honest validators that follow the protocol should receive rewards.
(2) Byzantine validators that deviate from protocol specifications should suffer from penalties.

Most studies known so far focus on the incentive flaws that trigger condition (1), i.e., consider the *fair share* as the rewards honest validators should receive in the failure-free case, an attack makes honest validators receive a lower reward than their fair share or even suffer from penalties. For example, in the ex-ante reorg attack [4], the proposer will not receive its block rewards, and the attestors *lose* some attestation rewards. In the staircase attack [6], the adversary makes half of the randomly chosen honest validators suffer from penalties. Accordingly, we should also consider whether condition (2) is triggered and possibly the following cases:

- If honest validators receive lower rewards than their fair share, do Byzantine validators also receive lower rewards than their fair share or even suffer from penalties?
- If honest validators suffer from penalties, will Byzantine validators suffer from even higher penalties?

Inspired by the definition of *risk-free* attacks [31], we can assess the attack instances. Let fs be the fair share (of rewards) of an honest validator. Let R_B and R_h be the net pay-offs of Byzantine and honest validators, respectively. Positive pay-off denotes a *net reward*; negative pay-off denotes a *net penalty*. An attack is *meaningful* whenever $R_h < \mathsf{fs}$, i.e. honest validators earn less than their fair share. Based on (R_B, R_h), we classify each instance as follows:

Metric I: Both Byzantine and honest validators receive rewards. In this case, the attack is *meaningful* only when the rewards honest validators receive are lower than their fair share. Namely, $R_B > 0$ and $R_h > 0$ with $R_h < \mathsf{fs}$.

Metric II: Both Byzantine and honest validators are penalized. In this case, an attack is already meaningful, as honest validators are penalized while they strictly follow the protocol. Namely, $R_B < 0$ and $R_h < 0$.

Metric III: Byzantine validators receive rewards, while honest actions are penalized. An attack in this category may create a significant impact on the stability of the system, as the fraction of stake controlled by the adversary continues to grow and may eventually exceed the 33.3% threshold. Namely, $R_B > 0$ and $R_h < 0$.

Metric IV: Byzantine validators are penalized, while honest validators receive rewards. Similarly, the attack is *meaningful* only when the rewards honest validators receive are lower than their fair share. Namely, $R_B < 0$ and $R_h > 0$ with $R_h < \mathsf{fs}$.

To quantify the attack's impact, we introduce the *Byzantine Advantage* as $BA = R_B - R_h$. A larger BA means that the attacker's net pay-off is higher while the honest validators' net pay-off is lower. An attack is more *effective* if it has a larger BA. An attack is more *profitable* if $R_B - \mathsf{fs}$ is larger.

SO: Optimizing the attack using reinforcement learning. After identifying an incentive flaw, it is important to determine whether by slightly modifying the attack strategies and fine-tuning the attack parameters, a more profitable attack can be found. In our work, we use reinforcement learning (RL) to build SO. RL is well known for optimizing the long-term rewards through trials. The main challenge we address is that the reward of a validator may change while the system is running. Thus, it is hard to directly determine the "reward function", a crucial component for RL. To solve this problem, we choose a delayed-reward reinforcement learning model and simulate the rewards of validators while running the RL algorithm.

As we later show in the paper, using our RL-based SO, we are able to optimize two of our newly identified attacks: the selfish mining attack and the staircase attack-II. For staircase attack-II, the attack becomes significantly more effective with the optimization. The adversary can gain nearly twice the profit compared to the default attack (output of the SA component). In addition, the probability of launching the attack also improves from 35.0% to 97.0%.

IV. The BunnyFinder Framework

We now present the details of our BunnyFinder framework.

A. The Strategy Generator (SG)

As mentioned in Sec. III, the SG automatically generates some attack strategies. The basic strategies include order manipulation and content manipulation. For order manipulation, we use a *message queue* to denote the messages each validator is supposed to send. An adversary may modify its message queue arbitrarily and determine when each message is sent to other validators. The adversary has no control over the message queue by honest validators.

```
struct Block {
  slot:         unsigned int64,
  proposer:     unsigned int64,
  parent:       Hash,                // Can modify
  transactions: Vec<Transaction>,
  attestations: Vec<Attestation>     // Can modify
}

struct Attestation {
  slot:     unsigned int64,
  attestor: unsigned int64,
  head:     Hash,                    // Can modify
  source:   Checkpoint,              // Can modify
  target:   Checkpoint               // Can modify
}
```

Figure 3: The data structures of block and attestation.

For content manipulation, we have further identified several fields that might generate effective attack instances. We show the data structure of a block and an attestation in detail in Figure 3. We perform the following optimizations for content manipulation.

- (Block) The *slot* field denotes the slot number. The *proposer* field denotes the identity of the block proposer. As all validators can verify the identity of the block proposer, these two fields can not be modified. The *parent* field denotes the hash of the parent block. This field can be modified, a trick used by several known attacks [4], [6], [11], [40]. The *transactions* field includes a batch of transactions. This field is irrelevant to the attestation incentives. The *attestations* field includes a set of attestations. This field is crucial as each validator determines the *weight* of the branches in its block tree in the fork choice. To summarize, we consider modifying both parent and attestation fields for a block.
- (Attestation) Similar to the discussion for the block, we consider modifying slot and attestor fields meaningless for an attestation. The *head* field is the hash of each attestor's canonical chain. Meanwhile, the *source* and *target* fields are two checkpoints. These three fields can be modified, a practice conducted by several known attacks [6], [41].

An example. We use the ex-ante reorg attack [4] as an example to explain the attack strategies. The attack strategies are summarized in Figure 4. We summarize the message queues of the validators when no attack is launched in Figure 6a and the message queues when the attack is launched in Figure 6b. The legend shown in Figure 5 applies to Figure 6 and all following figures.

- *Order Manipulation in Slot 2*: As a proposer, the attacker delays its block for 3 units.
- *Content Manipulation in Slot 2*: As an attestor, the attacker sets the *head* field of the attestation to the hash of a withheld block.

Figure 4: Attack strategies in the ex-ante reorg attack.

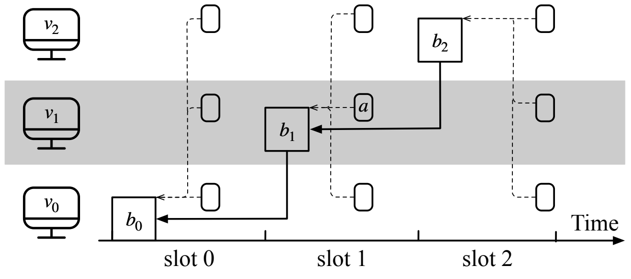
In this attack, only one proposer needs to be Byzantine but all Byzantine validators modify and delay their attestations. Both order manipulation and content manipulation are needed. All Byzantine attestors perform the same strategy.

Icons	Description
□	The block from an honest validator
🟥	The block from a Byzantine validator
□□	The attestations from honest validators
🟥🟥	The attestations from Byzantine validators
←	An action that points a block to its parent block
⇠	The action of an attestation voting for a block

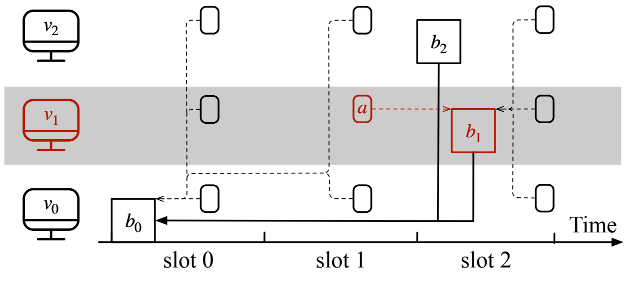
Figure 5: The legend of all figures.

Implementation details. In our implementation, we define a *strategy specification format* (SSF) in the form of JSON, specifying the exact time each message should be delayed and what content the message should be modified for the Byzantine validators. The SSF file is defined as some JSON templates for both order manipulation and content manipulation. Based on the concrete attack strategies (e.g., the number of units each message should be delayed), the corresponding template is "filled". To be specific, in each slot, our SSF defines three factors: `slot` denotes the slot Byzantine validators should take the actions, and `actions` denotes the attack strategies. One SSF file contains attack strategies for all slots in one epoch.

An example of a particular slot (e.g., slot 2) for the ex-ante reorg attack for a Byzantium validator is shown in Figure 7. In this example, `AttestBeforeSign` denotes the



(a) The message queues of three validators when no attacks are launched. Validators v_0, v_1, v_2 all behave honestly in this case. Each block is proposed on time and all honest attestors vote for these blocks in each slot.



(b) The message queues of three validators when the ex-ante reorg attack is launched. Validator v_1 is Byzantine and applies the attack strategies in Figure 4. Block b_1 is delayed and released at the beginning of slot 2. All Byzantine validators modify the *head* field of their attestation a as b_1, although their fork choice rule outputs b_2. After b_1 is launched, b_2 is re-organized.

Figure 6: The message queues of three validators in the failure-free case and under the ex-ante reorg attack [4]. Validator v_0, v_1, and v_2 are eligible proposers for block b_0 in slot 0, block b_1 in slot 1, and block b_2 in 2, respectively.

injection point that controls the *head* field of the attestation. Here, `modifyAttestHead:2` means that *head* field of the attestation in slot 2 is set as the hash of a block withheld in slot 2. In addition, `BlockBeforeBroadcast` denotes the injection point used to delay sending the block. `delayWithDuration:3` means that the block in slot 2 is withheld for three *units*, i.e., 12 seconds.

```
{
  "slots": [{
    "slot": "2",
    "actions": {
      "AttestBeforeSign":"modifyAttestHead:2",
      "BlockBeforeBroadcast":"delayWithDuration:3",
    }
  }]
}
```

Figure 7: An example of the JSON file that denotes the strategies in Figure 4. Some JSON fields denote modifying the fields of block/attestation shown in Figure 3. For example, modifyAttestHead means modifying the "attestation head" field of the attestation.

Our SSF can be easily extended. For instance, if new injection points are needed, more JSON fields can be added to SSF to connect SG with other modules. We provide more examples in Appendix B.

B. The Strategy Executor

The *Strategy Executor* mainly consists of two components: *Attacking Scheduler* (AS) and *Byzantine Simulator* (BS). AS contains two components: *Strategy Translator* (ST) and *Instruction Dispatcher* (ID). The ST parses the attack instances (generated by SG) written in JSON according to our SSF, and *translates* the corresponding attack strategies into executable code (instructions) in the codebase of the Ethereum implementation. ID interacts with the BS to further instruct the BS during the attack. The BS defines a set of injection points in its client, retrieves the corresponding instructions from the ID at runtime, executes these instructions at the injection points, and broadcasts the execution results to the network. We also build a *strategy pool* to store executable code so the complex attack strategies can be launched according to the instructions. We discuss more implementation details in our full version.

A running example (Figure 8). The process of attack injection proceeds as follows: ① When the probe of BS is triggered (at the `.BlockBeforeBroadcast` position), it sends a request to the *Instruction Dispatcher* (ID). The current thread of the Byzantine Simulator is set as the *suspended* state. ② Upon receiving further instruction, ID first parses the request and then looks up the strategy pool to determine if a local instruction is needed for `BlockBeforeBroadcast`. There are two possible outcomes:

- If no instruction is needed, the ID directly returns the remote instruction `CMD_NULL` to the BS.
- If an instruction is found, the ID executes the local instruction (e.g., `delayWithDuration`) and returns the remote instruction to BS.

After the instruction is executed, ③ BS receives the instruction response from the ID. ④ BS performs the remote instruction, and the suspended thread is resumed.

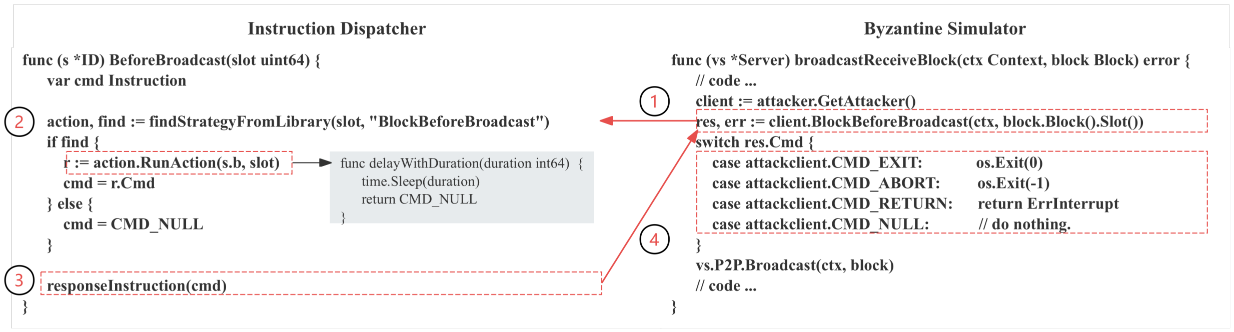
C. The State Analyzer

As mentioned in Sec. III-B, the *State Analyzer* (SA) analyzes the execution results and determines whether the corresponding attack is an incentive flaw. We discuss the implementation details in our full version.

D. The Strategy Optimizer

We use reinforcement learning to instantiate the strategy optimization module. We are particularly interested in fine-tuning the attack strategies to improve the profit of the attacks.

Review of reinforcement learning. Reinforcement Learning (RL) is a machine learning paradigm where an *agent* interacts with an *environment* over a sequence of steps (each step is numbered by t) to learn behaviors that maximize *cumulative reward* [26]. Formally, an RL problem is modeled as a Markov Decision Process (MDP), defined by the tuple (S, A, P, R), where S is the set of all possible states $S = \{s_1, \cdots\}$, A is the set of actions available to the agent $A = \{a_1, \cdots\}$, P is the transition function, and R is the reward function. The policy function is denoted as $s_{t+1} = P(s_t, a_t)$. The reward function is denoted as $r_{t+1} = R(a_t, r_t)$, where r is the reward.



```
func (s *ID) BeforeBroadcast(slot uint64) {
    var cmd Instruction

    action, find := findStrategyFromLibrary(slot, "BlockBeforeBroadcast")
    if find {
        r := action.RunAction(s.b, slot)
        cmd = r.Cmd
    } else {
        cmd = CMD_NULL
    }

    responseInstruction(cmd)
}
```

```
func delayWithDuration(duration int64)  {
    time.Sleep(duration)
    return CMD_NULL
}
```

```
func (vs *Server) broadcastReceiveBlock(ctx Context, block Block) error {
    // code ...
    client := attacker.GetAttacker()
    res, err := client.BlockBeforeBroadcast(ctx, block.Block().Slot())
    switch res.Cmd {
      case attackclient.CMD_EXIT:        os.Exit(0)
      case attackclient.CMD_ABORT:       os.Exit(-1)
      case attackclient.CMD_RETURN:      return ErrInterrupt
      case attackclient.CMD_NULL:        // do nothing.
    }
    vs.P2P.Broadcast(ctx, block)
    // code ...
}
```

Figure 8: Example of injection apply workflow.

The agent is the learner, also known as the decision maker. It observes the environment, selects actions, and adapts its behavior based on the outcomes of those actions. The agent's goal is to discover a policy P that maximizes its cumulative reward over time.

The environment interacts with the agent. The environment keeps track of the state of the "world", including the process, previous history, and restrictions of the world. In each step of the algorithm, the environment receives the agent's action, updates its internal state, and returns a new observation of the world that can be seen from the agent and the reward.

The state is a representation of the current situation faced by the agent. The state can be fully observable (as in standard Markov Decision Processes), or only partially observable by the environment, in which case the agent must infer hidden information using its observation history or memory.

The action is the decision made by the agent at each step. Actions influence how the environment evolves, and the agent must learn which actions lead to favorable outcomes over time.

Workflow of reinforcement learning. In each step t, the agent observes a state s_t from the environment, selects an action a_t from a predefined action space, and receives a scalar reward r_t (see Figure 9). The environment then transitions to a new state s_{t+1} according to its (possibly stochastic) dynamics. The agent's objective is to learn a policy that maximizes the expected cumulative reward over time.

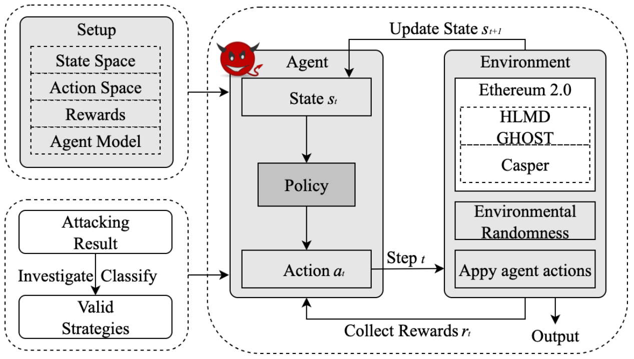


Figure 9: Schematic of our strategy optimizer.

Problem formulation. We formulate our problem as a delayed-reward reinforcement learning task and use a recurrent actor-critic model [42] trained with Proximal Policy Optimization (PPO) [43]. The choice is motivated by the sequential and partially observable nature of our environment. In addition, our reward function is defined as the difference between the reward of the honest validator and its fair share.

As mentioned previously, in the Ethereum PoS protocol, the reward of a validator changes over time, and can only be determined after some blocks are finalized. The immediate feedback would be misleading or even counterproductive. We thus cannot compute the rewards in each step. We therefore use the actor-critic framework. Namely, the actor learns a policy for selecting actions, while the critic estimates the value of states to support advantage-based learning. We augment the architecture with a recurrent memory module (LSTM) [44] to enable temporal credit assignment, i.e., the ability to connect actions with the outcomes that can only be observed in the future. The LSTM maintains temporal context over multi-slot attack sequences, tracking proposer duties, chain weights, and justification progression. PPO further stabilizes training by constraining policy updates and supporting efficient learning.

Agent and Environment. In our system, we model the agent as the adversary that controls all Byzantine validators. The environment simulates a minimal Ethereum 2.0 blockchain system, implementing the core elements related to the rewards of the validators. Specifically, we are interested in components: the Hybrid LMD-GHOST fork choice rule and the Casper FFG finality gadget. Conceptually, the environment represents an Ethereum 2.0 network consisting of all honest validators, and defines the rules on how blocks are proposed, attestations are sent, and rewards are computed.

We abstract away the workflow of Ethereum PoS protocol to integrate with the reinforcement learning framework. First, we map the notion of *slots* (in Ethereum) to *steps* (in RL). The protocol is no longer time-driven, but action-driven. Each RL step corresponds to one complete slot. In each step, the environment receives an action from the agent (i.e., the adversarial validator), updates the internal state of the chain, and computes the outcome according to our specification above. Second, we introduce attack-specific modifications to the environment. These changes allow us to focus on how the modified attack strategies can improve the outcome.

The state and action space are defined according to each attack instance. We defer the discussion to Appendix D. To summarize, SO can improve the success rate of our selfish mining attack and the staircase attack-II. Additionally, SO can decrease the stake requirement of the adversary of our selfish mining attack and increase the profit of our staircase attack. The results are summarized in Sec. III.

Extending BunnyFinder to other chains. BunnyFinder can be extended to other blockchains in the *propose-vote* paradigm. Namely, the SG can be adapted by redefining the attack space, and the SE only requires updating injection points. Such changes require additional engineering efforts. The SA and SO are chain-agnostic and are more general.

V. The New Incentive Flaws

We run our BunnyFinder framework to find incentive flaws. We summarize our new findings in Table I.

Experimental setup. We conduct our experiments on AWS EC2, using up to three c5.4xlarge instances (each with 16 vCPU and 32 GB memory). We deploy a local Ethereum 2.0 testnet to evaluate Byzantine behaviors within a controlled environment. Our codes are publicly accessible[3]. Details of our experimental setup can be found in Appendix C.

Details about the evaluation and validation process. Using our SG, we generated 9,354 attack instances. Our SE executed all attack instances, and the experimentation lasted for 145 hours. To identify whether each attack is an incentive flaw, we extract the execution logs and query them via SQL queries. Specifically, we collect the incentives received by the validators and classify them according to the metrics. Among all 9,354 attack instances, 3,121 of them can be identified as incentive flaws according to our metrics. We summarize the incentive flaws in Table II.

To further assess the attack instances, for each metric category, we identify the top 20% most profitable instances and manually analyze their corresponding attack strategies. If multiple instances exhibit similar behavior, such as differing only in delay length or minor parameter variations, we group them into the same class. For each class, we select the most effective strategy and apply RL as mentioned in our strategy optimizer (SO). The optimized strategies are re-evaluated via SE and SA. If the optimized attack instances belong to certain metrics, they are added to the corresponding category. In total, we have manually evaluated 300 attack instances and identified three new incentive flaws.

Overview of our results. By further classifying the attack instances, our framework successfully *reproduces* five known incentive attacks, including the ex-ante reorg attack [4], sandwich reorg attack [10], justification withholding attack [12], unrealized justification attack [11], and staircase attack [6]. We do not claim too much novelty in covering known attacks. Specifically, our SG is inspired by known attacks and use their basic attack strategies to generate the attack instances. It is thus not that surprising that our framework can cover known attacks. We have reproduced these attacks in both Prysm and Teku. Meanwhile, although some of the attacks are also covered by our framework, they have been mitigated in the recent *Deneb* upgrade of the codebase, i.e., post-Prysm version 5.0 and post-Teku version 24.2.0. We are able to reproduce these attacks on older versions.

We also find three *new* incentive attacks: selfish mining attack, staircase attack-II (i.e., a variant of the staircase attack), and pyrrhic victory attack. All the three attacks apply to both Prysm and Teku implementations. Finally and surprisingly, we find two implementation flaws in the *synchronization* module and *log* module in the Prysm implementation. These flaws do not appear in the Teku implementation.

Note that besides the fact that our tool can identify new incentive flaws, we consider our newly found attacks interesting, as many prior works focus on reporting new attacks [4]–[7] (based on manual work).

No.	Metrics	Count	Ratio	Attack Example
1	Metric-I	2586	82.86%	Selfish mining attack
2	Metric-III	2	0.64%	Staircase attack-II
3	Metric-II	466	17.07%	Pyrrhic victory attack
4	Metric-IV	67		

TABLE II: Summary of the incentive flaws.

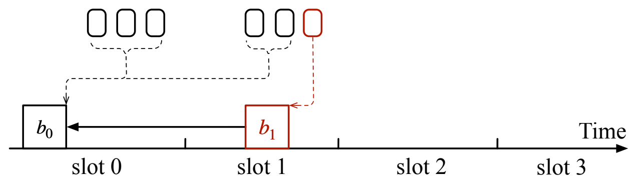
A. Selfish Mining Attack

Selfish mining is a well-known attack against Proof-of-Work (PoW) based blockchains (e.g., Bitcoin [45], Ethereum 1.0) [30]. In such an attack, a *mining pool* that controls a fraction of computational power may collude and gain more profit than its fair share. Recently, several works found that PoS also suffers from selfish mining [31]–[33]. A selfish mining attack is one type of reorganization attack where blocks from honest validators are discarded. However, a selfish mining attack has not been found for Ethereum PoS yet [31].

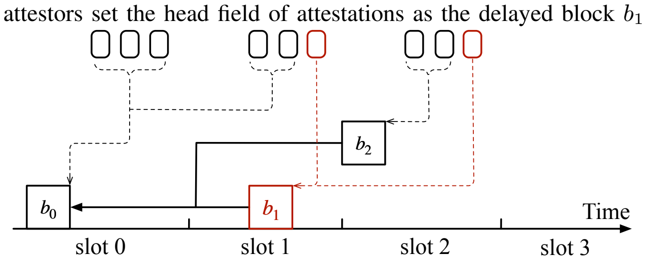
We identify the first selfish mining attack for Ethereum PoS. We show our JSON configuration file in Figure 11 and illustrate the attack in Figure 10. In this attack, the Byzantine validator withholds its blocks, strategically releasing them at an optimal time to maximize rewards. The delayed blocks accumulate a larger "weight" (i.e., a higher number of attestations) than the blocks produced by honest validators. Under the HLMD-GHOST fork choice rule [36], the delayed block replaces the honest validator's block in the canonical chain.

The effect of our selfish mining attack mirrors that of the original selfish mining attack: both Byzantine and honest validators receive rewards, but the rewards for honest validators are lower than their fair share. This attack aligns with *Metric I* in our criteria. As shown in Figure 12, the rewards for honest validators are lower than their fair share and the rewards

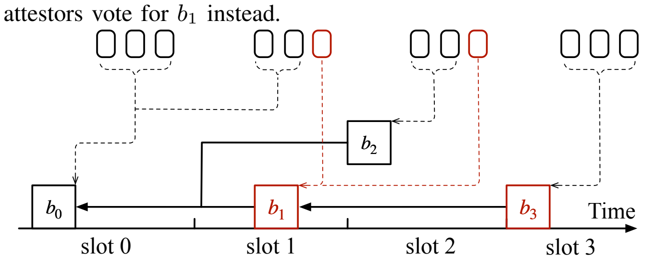
[3]Our codebase: https://doi.org/10.5281/zenodo.17042549



(a) In slot 0, the honest validator proposes a block b_0 and the attestors vote for b_0. In slot 1, the adversary delays its block b_1 for four seconds. Therefore, the honest attestors vote for b_0. Byzantine attestors set the head field of attestations as the delayed block b_1.



(b) In slot 2, the honest validators propose a block b_2 and tries to reorganize the delayed block b_1 due to the *honest reorg* mechanism [46]. In addition, honest attestors vote for b_2, while Byzantine attestors vote for b_1 instead.



(c) In slot 3, the adversary proposes a block b_3. The head of block b_3 is set as block b_1 instead of the output of fork choice, i.e., b_2. As the chain led by b_3 receives more attestations than the chain led by b_2, the chain led by b_3 becomes the canonical chain. Therefore, block b_2 is orphaned.

Figure 10: The message queues of three validators in the failure-free case and under our new selfish mining attack. The Byzantine validators apply the attack strategies in Figure 11.

```
{
  "slots": [{
    "slot": "1",
    "actions": {
      "AttestBeforeSign":"modifyAttestHead:1",
      "BlockBeforeBroadcast":"delayWithDuration:1",
    },
    "slot": "2",
    "actions": {
      "AttestBeforeSign":"modifyAttestHead:1",
    },
    "slot": "3",
    "actions": {
      "BlockGetNewParentRoot":"modifyParentRoot:1",
}}]}
```

Figure 11: An example of JSON file for selfish mining attack.

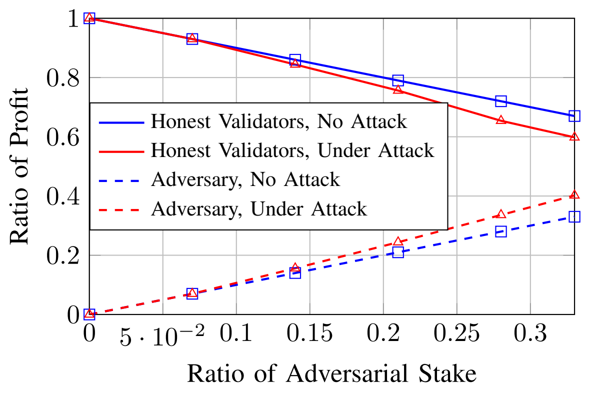


Figure 12: Profit of Byzantine validators and honest validators using the selfish mining strategy for different stake ratios, compared to the case without attack.

for Byzantine validators are higher than their fair share. In addition, the loss ratio depends on the stake owned by the adversary. The higher the stake owned, the lower the ratio of honest validators' profits.

Similar to the selfish mining attacks, the *profit* of the adversary grows as the fraction of stake controlled by the adversary grows, as shown in Figure 12.

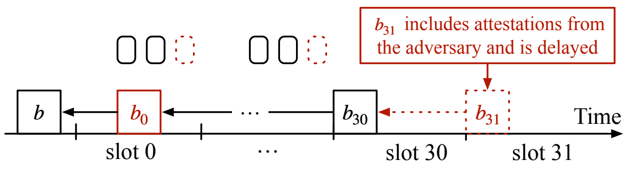
Note that, as summarized in Table II, most of our generated attack instances (that are incentive flaws by our metrics) belong to selfish mining. This is not surprising, as the adversary only needs to launch a reorganization attack to gain additional rewards. This is *easier* to launch than other attacks.

B. Staircase Attack-II

Staircase attack is a recent attack on the attestation incentives [6], as discussed in the introduction. The attack was mitigated via the *Deneb* upgrade, i.e., after Prysm version 5.0.

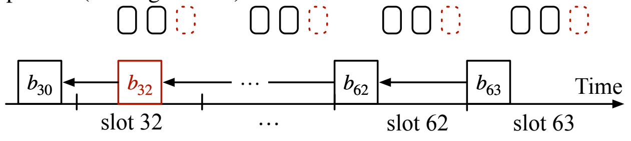
We find a new variant of the staircase attack, denoted as staircase attack-II. As shown in Figure 17, staircase attack-II requires the proposers of the first slot of two consecutive epochs to be Byzantine.

As shown in Figure 13, at the beginning of epoch 0, the adversary delays its block b_0 in slot 0 for four seconds. Therefore, the target of attestations from honest validators in slot 0, i.e., block b, is different from the target of attestations in slots 1-31, i.e., block b_0. In addition, all attestations from the adversary are delayed forever. Accordingly, the number of attestations from honest validators with the same target and source cannot reach $2n/3$. Block b_0, which is supposed to be justified in epoch 0, cannot be justified by the attestations from honest validators. In contrast, the adversary can justify the block in slot 0 by releasing its last block in epoch 0, i.e., block b_{31}. This block includes the attestations from the adversary, so the number of attestations exceeds two-thirds. The adversary delays the block for two epochs. As a result, the block in slot 0 can be justified at the end of epoch 2. In epoch 1, the adversary conducts the same strategies. Particularly, the adversary delays the block b_{32} for four seconds and delays all attestations from the adversary. Therefore, the honest validators cannot justify the block in epoch 1. After block b_{31} is released in epoch 2, block b_0 is justified. As the chain from honest validators does not justify any new block, the chain led by b_{31} is the new canonical chain. All attestations included in the chain from honest validators are discarded.

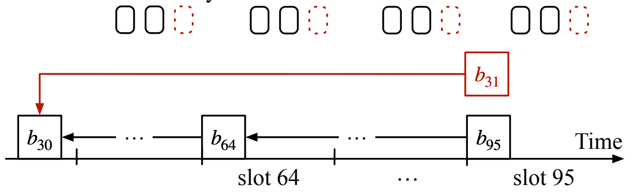


b_{31} includes attestations from the adversary and is delayed
b
b_0
b_{30}
b_{31}
Time
slot 0
slot 30
slot 31

(a) In epoch 0, the adversary delays its block in slot 0 for four seconds. In addition, all attestations from the adversary are delayed forever. In contrast, block b_{32} from the adversary in slot 31 includes all attestations from the adversary. b_{32} is delayed to the end of epoch 2 (see Figure 13c).



(b) In epoch 1, the adversary conduct similar strategies. In particular, the adversary delays its block in the first slot in epoch 1 for four seconds and delays its attestations forever.



(c) In epoch 2, the delayed block b_{31} is released at the end of epoch 2. After b_{31} is released, all blocks in epoch 1 and epoch 2 are reorganized and the attestations included are discarded.

Figure 13: The message queues of three validators in staircase attack-II. The Byzantine validators apply the attack strategies in Figure 17.

C. Pyrrhic Victory Attack

We find a new attack in which Byzantine validators suffer from penalties, but honest validators receive lower rewards or even suffer from penalties. As Byzantine validators also suffer from penalties, we call it a pyrrhic victory attack.

A basic pyrrhic victory attack is straightforward. For example, a pyrrhic victory attack can be launched based on the following strategy: All Byzantine validators delay their attestations every block forever (see JSON file in Figure 19). Without sending any attestations, all Byzantine validators suffer from penalties. Meanwhile, these actions also decrease the *participation rate* of the validators, as discussed in Sec. II. Accordingly, honest validators' rewards are lower than their fair share. This attack falls into metric IV in our criteria.

In the simple attack, the adversary suffers from a higher reward loss than the honest validators. In particular, the rewards of honest validators are approximately 67% of their fair share, while the loss rate of the adversary is about 180%. We do not consider this attack effective as the loss rate of the adversary is higher than that of honest validators.

We show the loss rates of both the adversary and honest validators in Figure 15a. The attack instances above the dashed line are all *effective*, as the loss rate of the adversary is lower than that of honest validators. The most effective one is highlighted, where the adversary loses 5.1% of its incentive awards to make honest validators suffer from 19.9% loss.

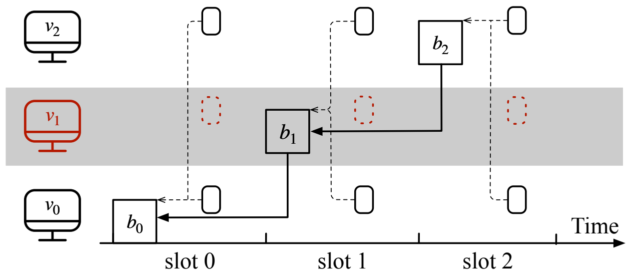


Figure 14: The message queues of three validators when the basic pyrrhic victory attack is launched. Validator v_1 is Byzantine and applies the attack strategies in Figure 19. All attestations from the adversary are withheld forever.

Further experimentation and analysis. Based on the basic strategies of the most effective attack instance we identified, i.e., the highlighted point in Figure 15a, we further fine-tune the concrete attack parameters to explore more *profitable* attack strategies. For example, we increase the delay time for Byzantine validators. Accordingly, we generate another 2,010 new *optimized* attack instances. As shown in Figure 15b, the number of effective attacks has increased from 60.78% to 70.78% after the optimization.

We now discuss one effective example. As shown in Figure 18, besides the attestations in slot 0 and slot 1, all attestations from the adversary are delayed forever. In addition, the proposers for blocks from slot 31 and slot 32 are Byzantine. Both b_{31} and b_{32} are withheld and released after four seconds have elapsed in slot 32. Before b_{31} and b_{32} are released, only the attestations from slot 0 to slot 30 are included in the canonical chain. Let the source of the attestations be *genesis block* and the target be b_0. After b_{31} and b_{32} are released, the attestations in slot 32 have the wrong source and target, since the correct source of the attestations should be b_0 and the target be b_{32}. Therefore, the corresponding validators suffer from attestation penalties. As the attestations from Byzantine validators besides slot 0 are delayed forever, the adversary also suffers from penalties as its attestations are not released. This attack falls into metric II in our criteria.

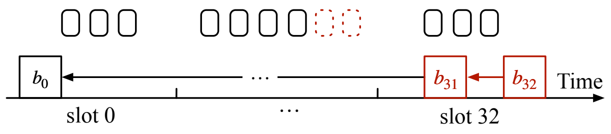
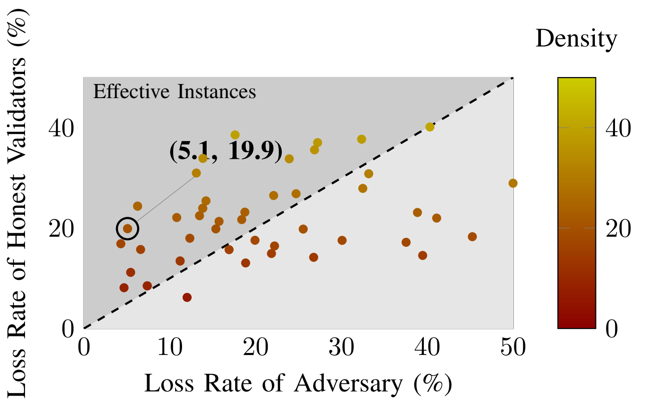
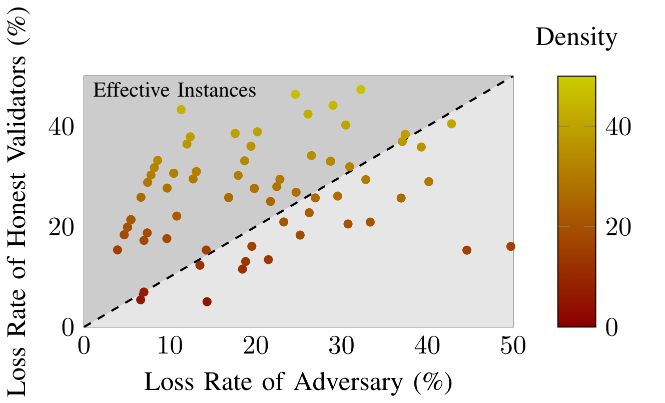


Figure 16: The message queues of an effective pyrrhic victory attack. The adversary applies the attack strategies in Figure 18. All attestations in slot 1 to slot 31 from the adversary are withheld forever. The adversary also delays block b_{31} in slot 31 and block b_{32} in slot 32 and releases them after four seconds have elapsed in slot 32. After b_{31} and b_{32} are released, the target and source of attestations from honest validators in slot 32 are set wrongly.



(a) Loss rate of adversary and honest validators before optimization.



(b) Loss rate of adversary and honest validators after optimization.

Figure 15: The loss rate of the adversary and honest validators under the Pyrrhic victory attack. The loss rate is calculated by dividing the number of reward losses by the fair share. The area above the dashed line represents *effective* attack instances where honest validators suffer from higher incentive loss than Byzantine validators. The ratio of effective attack instances increases after optimization.

D. Implementation Vulnerability

Synchronization issues. During one of our experiments, we encountered a deadlock issue for the *synchronization* module. In Ethereum, the synchronization module is used to *catch up* with the other validators when the validator realizes that it *falls behind*. Specifically, when a validator falls behind, it will not work until it finishes synchronization. When we launch the staircase attack-II, all validators stop querying the synchronization module. In particular, each validator maintains two local parameters: its *best* epoch (based on the head block) and others' best epoch (representing other validators' head blocks). When a validator's best epoch falls behind others' best epoch, it queries the synchronization module to catch up. Once the staircase attack-II is conducted, all honest validators believe that they fall behind and no validator can complete the synchronization.

After confirming the root cause, we believe this was an implementation vulnerability of the synchronization module in the Prysm codebase only (Golang implementation). Other implementations of Ethereum (i.e., Teku) do not suffer from such issues as block sync module is implemented differently.

Incorrect log. Another relatively minor finding we found is an issue due to logging. In particular, the reorg logger checks whether there is a reorganization in the canonical chain. However, due to the inconsistent invocation of the outdated functions, the reorg logger may be triggered in situations where no reorg has happened, creating non-existent reorg logs. While this does not affect the protocol, it does affect our state analyzer component.

VI. Optimizing the Attacks using Strategy Optimizer and Discussion

SO discovers strategies that significantly improve attack effectiveness compared to default approaches. Using our strategy optimizer, we are able to optimize both selfish mining attack and staircase attack-II. We summarize our findings in Table III. In this table, we compare the optimization results of the default strategy and the RL-optimized strategy, where the default strategy is defined as a staircase attack-II instance identified by our state analyzer. To evaluate the incentives, we define three metrics. The first metric is *Byzantine Advantage* (BA), which is defined as the difference between the average reward obtained by a Byzantine validator and that of an honest validator under the same strategy: $\text{BA} = \text{Byzantine Reward} - \text{Honest Reward}$.

This metric reflects the difference between the expected rewards in the attacks and the validators' fair share. A higher BA indicates that the attack is more profitable.

The second metric is *BA Rate*, which normalizes the advantage against a baseline (i.e., in our case, an incentive flaw generated by our SG):

$$\text{BA Rate} = \frac{\text{Byzantine Advantage}_{\text{RL-optimized}}}{\text{Byzantine Advantage}_{\text{Default}}}.$$

The third metric is *Attack Success Rate* (ASR). ASR measures the success rate of a strategy when applied to the system, where success is defined as the ability of the strategy to reduce the rewards of honest nodes.

As shown in Table III, the RL-optimized attack achieves a BA of 27.2. In contrast, the default attack achieves a BA of 13.4. The BA Rate is approximately 2.03, i.e., the RL-based optimization doubles the profit of the attack compared to the default attack. Meanwhile, the attack success rate increases from 35.0% to 97.0%. This indicates that the RL-optimized attack is significantly more effective than the default attack.

A. Selfish Mining

Due to space limitation, we present the details of using reinforcement learning to optimize the attack in Appendix D.

In the default selfish mining strategy generated by SA, the adversary adopts a static strategy: it tries to delay the block in slot t for 4 seconds and propose the block in slot $t+2$.

After the optimization by SO, the agent learns a more adaptive strategy. Namely, the agent dynamically adjusts its block release time based on whether Byzantine validators become proposers in the future. If so, the agent chooses to further withhold their blocks. In this way, the adversary can gain higher profit. Also, the agent dynamically changes the attack strategy when the stake controlled by the adversary is lower.

By applying SO, the optimized attack achieves a substantially higher success rate. In our experiments, the success rate is nearly 100%. In some experiments, the stake the adversary needs to own to launch the attack drops by over 20%.

Minimum requirement for selfish mining. After further manual analysis, we find that by controlling at least 13.4% validators, the attack is already profitable, i.e., Byzantine validators gain more rewards than their fair share, but honest validators receive lower rewards. We now explain why 13.4% is the threshold. Let β denote the ratio of the stake controlled by the adversary. As the validators are randomly assigned to 32 slots in an epoch, the total weight (informally, the maximum number of attestations) in a slot is $\frac{n}{32}$ and the ratio of the adversarial stake in a slot is $\frac{\beta}{32}$. As illustrated in Figure 10, after the adversary launches the attack, the weight of the chain led by block b_3 should be heavier than the weight of the chain led by block b_2. In particular, the chain led by block b_3 consists of the adversary's attestations from slots 2 and 3, i.e., $2 \times \frac{\beta}{32}$. In addition, block b_3 receives a *proposer boosting weight* [47], i.e., $0.4 \times \frac{n}{32}$. As a result, the weight of the chain led by b_3 is $\frac{2\beta}{32} + \frac{0.4n}{32}$. Meanwhile, the weight of the chain led by block b_2 consists of attestations from honest validators in slots 2 and 3, i.e., $2 \times \frac{(n-\beta)}{32}$. For the attack to succeed, we require: $\frac{2\beta}{32} + \frac{0.4n}{32} > \frac{2(n-\beta)}{32}$. By solving the inequality, we have $\beta > \frac{2}{15} \approx 13.4\%$.

The mitigation provided by Ethereum. We claimed in the introduction that our selfish mining attack has been mitigated in the recent update of the Prysm codebase, i.e., version v4.0.6 updated in May 2024[4]. In version v4.0.6, Ethereum *fixes* the problem using a `REORG_PARENT_WEIGHT_THRESHOLD` parameter that denotes whether attestations are released.

B. Staircase Attack-II

As summarized in Table III, the average rewards received by honest validators are significantly reduced and the average rewards received by the adversary are increased after the optimization. In extreme cases, all honest validators in several successive epochs are penalized and the adversary still receives its fair share.

TABLE III: Optimization improvement for staircase attack-II. *Default strategy refers to an attack instance that is identified as a staircase attack-II by our state analyzer.

Attack Strategy	Honest Validator	Byzantine validator	Byzantine Advantage	Attack Success Rate
Default*	-48.1	-34.7	13.4	35.0 %
RL-optimized method	-56.1	-28.9	27.2	97.0 %

Note: Byzantine Advantage is defined as the difference between the reward of the Byzantine node and the honest node. BA Rate is $27.2/13.4 \approx 2.02$.

[4]Prysm v4.0.6: https://github.com/prysmaticlabs/prysm/releases/tag/v4.0.6

In the default staircase attack-II generated by SA, the adversary strictly follows the same strategy. Namely, each Byzantine validator withholds all its attestations. This makes the adversary suffer from penalties if the withheld chain cannot reorganize the canonical chain later.

In contrast, after the optimization by SO, the agent learns a dynamic and state-aware strategy. In particular, instead of withholding every attestation and delaying its block for two epochs, the agent selectively chooses to withhold attestations only when the first slot of an epoch is controlled by the adversary. For instance, when the adversary does not control the first slot of two successive epochs, the agent will follow the protocol and will not withhold any attestations or delay blocks. The agent uses the proposer duty vector and justification state to determine its strategy. Also, the agent learns to conditionally withhold some of the attestations to gain more profit. Specifically, if an attestation is not included in the block, the attestor will receive only $5/7$ of its fair share. The agent thus strictly controls the attestations that are withheld.

With these improvements, the learned strategy might delay the justification of blocks for multiple epochs and improve the profit of the attack. The rewards by honest validators are up to 30% lower than those under the default strategy! Additionally, the cost of the adversary to conduct the attack decreases by 13%, dropping from 46.1% under the default strategy to 40.1% with the optimized policy.

Analysis. Our staircase attack-II shares the same feature as the staircase attack: Byzantine validators receive rewards while honest validators suffer from penalties, although honest validators strictly follow the specification of the protocol. Therefore, this attack falls into Metric III in our criteria. The probability of launching the attack is 1/9, as we require the proposer in the first slot to be Byzantine. Unlike the staircase attack, staircase attack-II cannot be continuously launched. However, if the adversary controls 33% stake and launches the attack whenever the attack can be launched, *all* honest validators will incur penalties for the corresponding epochs.

VII. Related Work

Penetration testing. Penetration testing is commonly used at the *later* stage of a software development cycle [15]–[18]. Informally, given an application (typically a web application), penetration testing allows ethical targeted attacks to be injected into the application to confirm whether there exist any design or implementation flaws. The targeted attacks can be generated manually via human experts or by automated tools, e.g., machine learning or large language models [16]–[18]. Similar to penetration testing, our framework injects some targeted attacks into the system. Differently, penetration testing usually injects the attacks at the *client* side, treating the application in a black-box manner. In contrast, our framework injects the attacks at Byzantine validators.

Fuzz testing. Fuzz testing aims to find software implementation vulnerabilities [21]–[23], [48], [49] by injecting non-deterministic testing approaches. In blockchain consensus, Tyr [21], LOKI [22], MPFUZZ [24] and CONFUZZIUS [50] are four recent fuzzing techniques. The idea is to trigger abnormal behavior (e.g., sending meaningless messages), sometimes at *all* nodes, to identify the software bugs. For instance, LOKI [22] finds some common vulnerabilities and exposures (CVEs) for Ethereum and Hyperledger Fabric. MP-FUZZ [24] finds new asymmetric-DoS vulnerabilities on six major Ethereum clients. In contrast, our workflow is inspired by fuzz testing that injects failures to the codebase to find uncovered issues. Compared to fuzz testing, our framework targets design flaws rather than implementation flaws. As a surprising finding, one of our attack instances also identifies an implementation flaw.

Byzantine simulator. Byzantine simulators are used for assessing the robustness of Byzantine fault-tolerant (BFT) protocols by simulating malicious or unpredictable behavior within the system. Twin [51] develops a unit testing tool that implements three Byzantine behaviors for assessing Byzantine fault-tolerant (BFT) protocols. The tool assesses a BFT-based system Diem [52], and tries to verify whether the implementation is correct. It was shown that the tool can also be extended to cover some known attacks on some BFT protocols. Similar to Twin, we also inject Byzantine behaviors. Unlike Twin, which mainly focuses on safety and liveness threats (which are not supposed to happen if the implementation matches the design of a provably secure protocol), our approach focuses on incentive flaws.

Chaos engineering. Chaos engineering [53] is an approach that assesses whether a system can still operate normally under *errors* [19], [20], [54]. The errors are often invalid system calls. By comparing the behavior of the system under and without the injected system calls, the *stability* (sometimes called resilience in the literature of chaos engineering) is evaluated. For example, CHAOSETH [19] conducts a chaos engineering approach to an Ethereum implementation and identifies potential issues that may make the validators crash under errors. In contrast, our approach does not aim to assess whether the system can still operate correctly under errors but rather to find incentive flaws.

RL-based tools. Several studies have employed reinforcement learning to solve vulnerabilities in blockchain systems [55]–[57]. The closest work of our strategy optimizer is one due to SquirRL [55]. SquirRL builds a reinforcement learning based framework to analyze selfish mining in Bitcoin and a simplified version of Ethereum. The idea is to analyze whether a Nash equilibrium can be reached under different adversarial models. In contrast, we use RL to improve the attack strategies.

Known attacks to Ethereum. Many attacks have revealed incentive flaws in Ethereum PoS. Zhang et al. proposed the staircase attack [6]. In the attack, an adversary controlling more than 29.6% of the stake can suppress attestation rewards for all honest validators. D'Amato et al. introduced the sandwich reorg attack [10]. This attack allows collusion between two Byzantine proposers to result in the orphaning of blocks proposed by honest validators, thereby depriving them of their rightful rewards. Potuz described the justification withholding attack [12]. In this attack, validators privately cast but withhold votes to delay finality. Asgaonkar et al. proposed the unrealized justification attack [11], where the adversary withholds a justified checkpoint to gain more rewards.

Recently, Nero showed that validators can manipulate the RANDAO (i.e., the randomness used to select proposers and attestors) to gain extra benefits such as rewards [58]. Our current work does not incorporate the strategies of RANDAO manipulation, but it can be integrated into BunnyFinder to possibly obtain new attacks. We leave it as future work.

Many attacks focus on goals beyond incentives. Yaish et al. analyzed speculative denial-of-service (DoS) attacks [59]. In this attack, adversaries can craft malicious transactions to clog blockchain actors' mempools, forcing victims to produce empty blocks. Li et al. introduced DETER mempool DoS attacks [60]. The vulnerabilities allow adversaries to use low-fee transactions to evict mempool transactions. Rodler et al. presented EVMPatch [61], a framework designed to automatically and instantly patch vulnerabilities in Ethereum smart contracts.

VIII. Conclusion

We present BunnyFinder, a framework designed to identify incentive flaws in the Ethereum protocol. Inspired by software testing technologies, BunnyFinder provides a framework that automatically injects generated failures into a simulated network, analyzes the incentives received by the validators, and determines the presence of attacks. We have simulated 9,354 attacks generated by our framework. Our results show that 32.9% of the attacks belong to incentive flaws. By further categorizing the flaws, our framework covers five known incentive attacks and uncovers three new attacks. As a by-product, it also reveals two implementation flaws.

Acknowledgment

We would like to thank Yu Jiang and Chao Zhang for their helpful discussion on software testing tools. This work was supported in part by the National Key R&D Program of China under 2023YFB2704300, the National Natural Science Foundation of China under 62502266, China Postdoctoral Science Foundation under 2023M741949, a research grant from the Ant Group, and Tsinghua Shuimu Scholar.

Ethical Considerations

Our experiments were conducted locally using open-source libraries and public datasets, without access to any external or live systems. These experiments do not involve animals, humans, the environment, healthcare, or military applications. We carefully followed the ethical principles of the Menlo Report throughout our experimental design.

A. Research Ethics Considerations

We are committed to complying with all relevant research ethics considerations. In particular, we are committed to the following principles:

- **Respect for Persons:** Our research does not involve human subjects or personal data. We respect the work of other researchers and properly cite all relevant prior work.
- **Beneficence:** Our findings highlight a critical incentive vulnerability in Ethereum.
- **Justice:** We strive to ensure our proposed modifications do not disproportionately impact or disadvantage any particular group. We have already disclosed our findings to Ethereum developers.
- **Respect for Law and Public Interest:** No actions have been taken to exploit the identified vulnerabilities; instead, our research was conducted with the goal of improving the Ethereum system.

REFERENCES

[1] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[2] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining ghost and casper," *arXiv preprint arXiv:2003.03052*, 2020.

[3] CoinMarketCap. (2025) Ethereum (eth) price, charts, and news. [Online]. Available: https://coinmarketcap.com/currencies/ethereum/

[4] M. Neuder, D. J. Moroz, R. Rao, and D. C. Parkes, "Low-cost attacks on ethereum 2.0 by sub-1/3 stakeholders," *arXiv preprint arXiv:2102.02247*, 2021.

[5] C. Schwarz-Schilling, J. Neu, B. Monnot, A. Asgaonkar, E. N. Tas, and D. Tse, "Three attacks on proof-of-stake ethereum," in *FC*, 2022, pp. 560–576.

[6] M. Zhang, R. Li, and S. Duan, "Max attestation matters: Making honest parties lose their incentives in ethereum pos," in *USENIX Security*, 2024.

[7] R. Nakamura, "Analysis of bouncing attack on ffg," https://ethresear.ch/t/analysis-of-bouncing-attack-on-ffg/6113, (accessed in Feb 2024).

[8] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 183–198.

[9] V. Buterin, "Discouragement attacks," 2018.

[10] F. D'Amato and C. Schwarz-Schilling, "Proposer boost considerations," https://notes.ethereum.org/@casparschwa/H1T0k7b85, (accessed in Feb 2024).

[11] A. Asgaonkar, "Unrealized justification reorgs," https://notes.ethereum.org/@adiasg/unrealized-justification, (accessed in Feb 2024).

[12] Potuz, "Justification widtholding attacks," https://hackmd.io/o9tGPQL2Q4iH3Mg7Mma9wQ, (accessed in Feb 2024).

[13] P. Chaidos, A. Kiayias, and E. Markakis, "Blockchain participation games," in *International Conference on Web and Internet Economics*. Springer, 2023, pp. 169–187.

[14] S. Motepalli and H.-A. Jacobsen, "Reward mechanism for blockchains using evolutionary game theory," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2021, pp. 217–224.

[15] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, 2005.

[16] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "PentestGPT: Evaluating and harnessing large language models for automated penetration testing," in *USENIX Security*, 2024, pp. 847–864.

[17] T. Lee, S. Wi, S. Lee, and S. Son, "Fuse: Finding file upload bugs via penetration testing." in *NDSS*, 2020.

[18] G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and D. Wang, "NAUTILUS: Automated RESTful API vulnerability detection," in *USENIX Security*, 2023, pp. 5593–5609.

[19] L. Zhang, J. Ron, B. Baudry, and M. Monperrus, "Chaos engineering of ethereum blockchain clients," *Distributed Ledger Technologies: Research and Practice*, 2023.

[20] S. Sondhi, S. Saad, K. Shi, M. Mamun, and I. Traore, "Chaos engineering for understanding consensus algorithms performance in permissioned blockchains," in *DASC/PiCom/CBDCom/CyberSciTech*, 2021.

[21] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 2517–2532.

[22] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "LOKI: State-aware fuzzing framework for the implementation of blockchain consensus protocols," in *NDSS*, 2023.

[23] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in Ethereum via multi-transaction differential fuzzing," in *OSDI*, 2021, pp. 349–365.

[24] Y. Wang, Y. Tang, K. Li, W. Ding, and Z. Yang, "Understanding ethereum mempool security under asymmetric DoS by symbolized stateful fuzzing," in *USENIX Security*, 2024, pp. 4747–4764.

[25] Gartner, "Breach and attack simulation (bas) tools," https://www.gartner.com/reviews/market/breach-and-attack-simulation-bas-tools, (accessed in Nov 2024).

[26] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *IEEE Trans. Neural Networks*, vol. 9, pp. 1054–1054, 1998. [Online]. Available: https://api.semanticscholar.org/CorpusID:60035920

[27] Offchain Labs, "Prysm consensus client v5.3.3," https://github.com/OffchainLabs/prysm/tree/v5.3.3, 2025.

[28] ConsenSys, "Teku client v25.6.0," https://github.com/Consensys/teku/tree/25.6.0, 2025.

[29] M. Zhang, R. Li, X. Lu, and S. Duan, "Available attestation: Towards a reorg-resilient solution for ethereum proof-of-stake," in *USENIX Security*, 2025.

[30] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Communications of the ACM*, vol. 61, no. 7, pp. 95–102, 2018.

[31] J. Brown-Cohen, A. Narayanan, A. Psomas, and S. M. Weinberg, "Formal barriers to longest-chain proof-of-stake protocols," in *Proceedings of the 2019 ACM Conference on Economics and Computation*, 2019, pp. 459–473.

[32] M. V. Ferreira and S. M. Weinberg, "Proof-of-stake mining games with perfect randomness," in *EC*, 2021, pp. 433–453.

[33] M. Neuder, D. J. Moroz, R. Rao, and D. C. Parkes, "Selfish behavior in the tezos proof-of-stake protocol," *arXiv preprint arXiv:1912.02954*, 2020.

[34] E. Budish, A. Lewis-Pye, and T. Roughgarden, "The economic limits of permissionless consensus," in *Proceedings of the 25th ACM Conference on Economics and Computation*, 2024, pp. 704–731.

[35] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[36] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *FC*. Springer, 2015, pp. 507–527.

[37] Y. Liu, Y. Lu, K. Nayak, F. Zhang, L. Zhang, and Y. Zhao, "Empirical analysis of eip-1559: Transaction fees, waiting times, and consensus security," in *CCS*, 2022, pp. 2099–2113.

[38] R. Pass and E. Shi, "Fruitchains: A fair blockchain," in *PODC*, 2017, pp. 315–324.

[39] Y. Huang, J. Tang, Q. Cong, A. Lim, and J. Xu, "Do the rich get richer? fairness analysis for blockchain incentives," in *SIGMOD*, 2021, pp. 790–803.

[40] J. Neu, E. N. Tas, and D. Tse, "Ebb-and-flow protocols: A resolution of the availability-finality dilemma," in *SP*, 2021, pp. 446–465.

[41] "Prevention of bouncing attack on ffg," https://ethresear.ch/t/prevention-of-bouncing-attack-on-ffg/6114, (accessed in Feb 2024).

[42] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Advances in neural information processing systems*, vol. 12, 1999.

[43] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[44] A. Graves and A. Graves, "Long short-term memory," *Supervised sequence labelling with recurrent neural networks*, pp. 37–45, 2012.

[45] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," *Journal of the ACM*, vol. 71, no. 4, pp. 1–49, 2024.

[46] M. Sproul, "Allow honest validators to reorg late blocks," https://github.com/ethereum/consensus-specs/pull/3034, (accessed in Augest 2024).

[47] "Proposer lmd score boosting #2730," https://github.com/ethereum/consensus-specs/pull/2730, (accessed in Feb 2024).

[48] A. Sorniotti, M. Weissbacher, and A. Kurmus, "Go or no go: Differential fuzzing of native and c libraries," in *2023 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2023, pp. 349–363.

[49] S. Sharma, S. R. Tanksalkar, S. Cherupattamoolayil, and A. Machiry, “Fuzzing api error handling behaviors using coverage guided fault injection,” in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1495–1509.
[50] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, “Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts,” in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 103–119.
[51] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, “Twins: Bft systems made robust,” in *PODC*, 2022.
[52] Diem, “Diembft,” https://github.com/diem/diem, (accessed in Nov 2024).
[53] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, “Chaos engineering,” *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.
[54] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, “Blockbench: A framework for analyzing private blockchains,” in *SIGMOD*, 2017.
[55] C. Hou, M. Zhou, Y. Ji, P. Daian, F. Tramèr, G. Fanti, and A. Juels, “Squirrl: Automating attack analysis on blockchain incentive mechanisms with deep reinforcement learning.”
[56] R. De Silva, W. Guo, N. Ruaro, I. Grishchenko, C. Kruegel, and G. Vigna, “{GuideEnricher}: Protecting the anonymity of ethereum mixing service users with deep reinforcement learning,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 3549–3566.
[57] W. Li, L. Xue, Z. Han, B. Chen, X. Zhang, and X. Zhou, “Autominer: Reinforcement learning-based mining attack simulator,” in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2024, pp. 222–241.
[58] Nero_eth, “Selfish mixing and randao manipulation,” Ethereum Research post, Jul. 2023, accessed July 2025. [Online]. Available: https://ethresear.ch/t/selfish-mixing-and-randao-manipulation/16081
[59] A. Yaish, K. Qin, L. Zhou, A. Zohar, and A. Gervais, “Speculative {Denial-of-Service} attacks in ethereum,” in *33rd USENIX security symposium (USENIX Security 24)*, 2024, pp. 3531–3548.
[60] K. Li, Y. Wang, and Y. Tang, “Deter: Denial of ethereum txpool services,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1645–1667.
[61] M. Rodler, W. Li, G. O. Karame, and L. Davi, “{EVMPatch}: Timely and automated patching of ethereum smart contracts,” in *30th usenix security symposium (USENIX Security 21)*, 2021, pp. 1289–1306.
[62] Ethresearch, “Selfish mining in pos,” https://ethresear.ch/t/selfish-mining-in-pos/15551, 2024, eTH Research.
[63] mart1i1n, “Staircase attack-ii in ethereum pos,” https://ethresear.ch/t/staircase-attack-ii-in-ethereum-pos/22099/10, Apr. 2025, ethereum Research Forum.
[64] Mart1i1n, “Deadlock in synchronization module under staircase attack-ii,” https://github.com/OffchainLabs/prysm/issues/15144, 2025, gitHub Issue.
[65] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[66] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” *arXiv preprint arXiv:1506.02438*, 2015.
[67] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

Appendix A
Current Status of Vulnerability

We summarize the current status of vulnerability found in Table IV. All of our findings are acknowledged by Ethereum.

Appendix B
SSF JSON File

Figures 17, 19, and 18 display JSON files in the Strategy Specification Format (SSF) for configuring attack scenarios within our BunnyFinder framework. Each JSON file defines specific actions at various slots, such as `exit`, `delayWithDuration`, and `modifyParentRoot`, to simulate targeted behaviors in attacks like staircase attack-II and two variants of the pyrrhic victory attack. These configurations enable precise manipulation of validator actions to analyze the impact on Ethereum PoS.

Vulnerability	**Reported**	**Acknowledged**	**Addressed**	**Channel.**
Selfish mining	Yes	Yes	Yes	CW [62]
Staircase attack-II	Yes	Yes	No	CW [63]
Pyrrhic Victory Attack	Yes	Yes	No	MT
Synchronization issues	Yes	Yes	No	CW [64]
Incorrect log	Yes	Yes	Yes	MT, EM

TABLE IV: Status of Discovered Vulnerabilities. “Reported” indicates submission to Ethereum; “Addressed” indicates a fix or mitigation; “Channel” indicates the form of disclosure, with CW indicating through community websites, MT indicating through meeting reports, and EM indicating email.

```
{
  "slots": [{
    "slot": "all",
    "actions": {
      "AttestBeforeBroadcast":"exit",
    },
    "slot": "0",
    "actions": {
      "BlockBeforeBroadcast":"delayWithDuration:1",
    },
    "slot": "31",
    "actions": {
      "BlockGetNewAttestations":"true",
      "BlockBeforeBroadcast":"delayWithDuration:192",
    },
    "slot": "32",
    "actions": {
      "BlockBeforeBroadcast":"delayWithDuration:1",
    },
  }]
}
```

Figure 17: The JSON file in SSF for the staircase attack-II.

```
{
  "slots": [{
    "slot": "1-31",
    "actions": {
      "AttestBeforeBroadcast":"exit",
    },
    "slot": "31",
    "actions": {
      "BlockBeforeBroadcast":"delayWithDuration:4",
    },
    "slot": "32",
    "actions": {
      "BlockGetNewParentRoot":"modifyParentRoot:31",
      "BlockBeforeBroadcast":"delayWithDuration:1",
    },
  }]
}
```

Figure 18: The JSON file in SSF for the second variant of pyrrhic victory attack.

```
{
  "slots": [{
    "slot": "all",
    "actions": {
      "AttestBeforeBroadcast":"exit",
    }
  }]
}
```

Figure 19: The JSON file in SSF for the first variant of pyrrhic victory attack. The action `exit` means that the function of broadcasting an attestation is directly terminated.

Appendix C
Evaluation Details

Experimental setup. There are three types of nodes: beacon nodes, execution nodes, and validator nodes. The beacon nodes establish a peer-to-peer network (called the consensus layer) via the "libp2p" protocol[5]. Execution nodes establish a peer-to-peer network (called the execution layer) via the "devp2p" protocol[6]. The beacon nodes communicate with the validator nodes and the execution nodes via the "gRPC" protocol.

We show an example of the network topology in Figure 20. This topology has two layers and includes beacon nodes, execution nodes, and validator nodes. In our testnet, we require some beacon nodes (highlighted in red in the figure) to be modified to act as Byzantine nodes. This modification introduces adversarial behavior while maintaining connections with other nodes in the network.

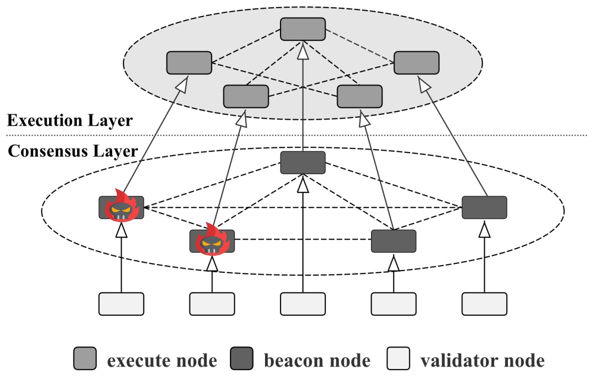


Figure 20: Overview of BunnyFinder network topology.

Experimental optimization. As discussed in Sec. III, each slot allows modification of five data structures: `parent`, `attestations`, `head`, `source`, and `target`, and supports two message types to delay: block and attestation. Any combination of at least 7 strategies can be selected per slot, with the total combinations given by $\sum_{k=0}^{n} C_n^k = 2^n$. Thus, each slot has $2^7 = 128$ possible strategies. One attacking instance requires 3 epochs (96 slots) to launch and we can generate up to 128^{96} theoretical strategy combinations. Under the current configuration, completing one instance takes approximately 19 minutes, making it impractical to explore all combinations. To improve efficiency, we reduced the slot time from 12 seconds to 3 seconds, cutting the average time per experiment from 19 minutes to 4.8 minutes without compromising result correctness.

[5]libp2p: https://blog.ipfs.tech/2020-06-09-libp2p-in-2020/

[6]devp2p: https://geth.ethereum.org/docs/tools/devp2p

Appendix D
Reinforcement Learning Details

In this section, we provide detailed implementations of our reinforcement learning approach.

A. *Environment*

We summarize the environmental changes as follows:

- (Selfish mining) In this attack, the goal of the adversary is to manipulate the fork choice rule. Finality mechanisms, i.e., Casper FFG, are not directly involved, as the attack does not aim to reverse finalized blocks. Thus, we remove the Casper FFG in the environment. We adopt a simplified finality heuristic where a block is considered finalized if it remains on the canonical chain for eight slots. We use eight slots because we observe that most reorganizations in selfish mining attacks cannot last for longer than eight slots. Rewards are computed based on this finality approximation.
- (Staircase attack-II) The attack aims to delay the justification of blocks and manipulate the canonical chain by honest validators. The attack strategies do not involve modifying the weight of the fork choice. We simplify the LMD-GHOST logic by only keeping the pruning mechanism. Furthermore, as the action does not involve a temporary blockchain fork, we modify each action taken by the agent as an epoch.

B. *State and Action Space*

A key component of the state space is the agent's proposer duty vector, which encodes whether the Byzantine validator is scheduled to propose blocks in a fixed future time window. Formally, this is represented as a binary vector of length t, i.e., $\texttt{duty} \in 0, 1^t$, where each entry indicates whether the agent is the designated proposer in the corresponding slot. According to the current Ethereum system, the information is known in advance. Proposer's duty influences the agent's planning horizon: since the effectiveness of certain attacks depends on when the adversary controls the proposer, having visibility into upcoming proposer slots makes it easy to adjust the attack strategies.

- (Selfish mining) We define the state observed from the environment as $s_t^{\text{selfish}} = [duty, w_{\text{pub}}, w_{\text{priv}}]$. Proposer duty vector $duty \in \{0, 1\}^8$ indicates whether the adversary is the block proposer in the next eight slots. Since the attack is closed related to the slots with Byzantine proposers, this state determines whether the agent should propose a block or withhold the block. It also defines the "attack window", successful selfish mining hinges on well-timed block production by the adversary. Let $w_{\text{pub}} \in \mathbb{N}$ represents the weight of the public chain. This weight reflects the accumulative attestations of all validators (including honest ones) and is the input to fork choice. For the attack to succeed, the private chain must eventually exceed this weight. Let $w_{\text{priv}} \in \mathbb{N}$ be the weight of the private chain. This value tracks the adversary's progress in building the private chain. It informs the agent when the attack becomes profitable.

- (Staircase attack-II) Let $s_t^{\text{staircase}} = [duty, j_{\text{pub}}, j_{\text{priv}}, j_{\text{global}}]$ be the state. As in selfish mining, the proposer duty vector $duty \in \{0,1\}^{64}$ indicates whether the adversary controls the proposer in the current epoch and the next epoch. Let j_{pub} be the last justified checkpoint on the public chain. The adversary must monitor j_{pub} to adjust its strategy. Let j_{priv} be the last justified checkpoint of the private chain. This discrepancy of j_{pub} and j_{priv} forms the basis of the staircase attack, as mentioned in Sec. V.

The agent's behavior consists of two decisions: a block proposal strategy and an attesting strategy. Based on the targets of the two attacks, we use the following actions in the reinforcement learning:

- (Selfish mining) There are three actions for each Byzantine proposer: propose a block according to the protocol, withhold the block, and release previously withheld block(s). The first action is the behavior of an honest validator, so taking this action does not yield a higher reward than the fair share. The second and third actions are related to the profit of the attack. Additionally, there are two attesting actions: send the attestation according to the protocol; withhold the attestation and release it later. Similar to the proposal actions, the second action is related to the profit of the attack.
- (Staircase attack-II) The actions in the staircase attack reinforcement learning model are $\mathcal{A}_{\text{staircase}} = [p_1, p_2, p_3, a_1]$, where $p_1 \in \{0,1\}$, $p_2 \in \{0,1\}$, and $p_3 \in \{0,1\}$ are proposal actions and $a_1 \in \{0,1\}$ is attesting action. The p_1 action determines whether the block is delayed. If so, the block is delayed by four seconds. This disrupts the start of the voting period and reduces the chances of successful justification. The action p_2 controls whether the last Byzantine proposer delays its block. If so, the agent will include all attestations from Byzantine validators in its block and withhold it. The private chain will justify a higher checkpoint. The action p_3 determines whether to release the withheld blocks. The action a_1 controls whether the adversary withholds their attestations. Withholding the attestations might disrupt justification of the public chain.

C. Reward Function

We design tailored reward functions for different attack types to guide the RL agent toward behaviors that maximize attack effectiveness.

- (Selfish mining) For selfish mining attacks, we focus on causing block reorganizations. The reward function is defined as $R_{selfish}(t) = \alpha \cdot N_{reorg}(t) + \beta \cdot (W_{adv}(t) - W_{honest}(t))$, where $N_{reorg}(t)$ is the number of honest blocks reorganized at slot t, W_{adv} and W_{honest} are the cumulative weights of adversarial and honest blocks respectively, and α, β are weighting coefficients.
- (Staircase attack-II) For staircase attack-II, we aim to discard honest attestations through chain reorganizations. The reward function is defined as $R_{staircase}(t) = \delta \cdot N_{discarded}(t) + \epsilon \cdot (R_{adv}(t) - R_{honest}(t))$, where $N_{discarded}$ is the number of honest attestations discarded from the canonical chain, R_{adv} and R_{honest} are rewards received by adversarial and honest validators, and δ, ϵ are weighting coefficients.

Given the multi-epoch nature of attacks, we use delayed rewards $R_{total}(t) = \Sigma_{i=0}^{\tau} \gamma^i \cdot R(t+i)$, where τ is the delay time, γ is the discount factor, and R is the current rewards at slot t. All rewards are normalized relative to honest protocol baselines to ensure consistent learning across different stake distributions.

D. Architecture of the Model

We adopt a recurrent actor-critic architecture to serve as the agent's predictive model, enabling memory-dependent decision-making under partial observability and delayed feedback. The model is structured around three core components: a feedforward feature encoder, a recurrent memory module, and two output heads for action selection and value estimation.

In each step, the agent receives a structured state vector that includes current chain features and the proposer duty vector. This input is first passed through a feedforward encoder, i.e., a multi-layer perceptron (MLP) [65] with one hidden layer of 512 units and ReLU activation, which transforms the raw state into a fixed-dimensional embedding. This encoded representation is then processed by a two-layer Long Short-Term Memory (LSTM) network, each layer containing 256 hidden units. The LSTM maintains temporal context over time, allowing the agent to track multi-step attack dynamics, such as private chain length, justification progression, or delayed proposal release schedules. The recurrent design is essential for capturing patterns in environments with partial observability and delayed reward signals, where the consequences of an action may unfold over several future steps.

The output of the LSTM is shared between two separate linear heads. The actor head outputs unnormalized logits over the discrete action space, which are converted into action probabilities via a softmax function. These probabilities define the agent's stochastic policy, from which actions are sampled during training and inference. The critic head produces a scalar value estimate of the expected return from the current state, which is used to compute advantage estimates during training. Both outputs are optimized jointly.

The model is trained end-to-end using Proximal Policy Optimization (PPO), with Generalized Advantage Estimation (GAE) [66] to smooth temporal credit assignment. The PPO loss consists of a clipped policy gradient term for stable updates, a value regression term to train the critic, and an entropy bonus to encourage exploration. We use the Adam optimizer [67] with a learning rate of 3×10^{-4}, gradient clipping (max norm 0.5), and run updates over minibatches sampled from trajectories collected in parallel environments.

Appendix E
Artifact Appendix

A. Description & Requirements

We present BunnyFinder, a framework for finding incentive flaws in Ethereum. Our artifact consists of the BunnyFinder implementation.

1) How to access: The artifact can be accessed by downloading from Zenodo. All the scripts, container images, source codes, and sample output files can be accessed via the URL: https://doi.org/10.5281/zenodo.17042549.

2) Hardware dependencies: The experiments do not require any specialized hardware. Our test environment is a computer with an 8-core CPU, 16 GB of RAM, 100 GB of storage, and a 100 Mbps network connection.

3) Software dependencies: Our experiments require a host running Ubuntu 22.04 or higher, with Docker installed according to the official documentation (Engine version 24.0.6 or higher), plus the docker-compose plugin and the Kurtosis framework.

4) Benchmarks: None

B. Artifact Installation & Configuration

After installing Docker, run the following steps:

1. Download the repository zip file and unzip[7]:

```
curl -o bunnyfinder.zip https://zenodo.org/
records/17042549/files/bf_workspace.zip?download=1
&& unzip bunnyfinder.zip
```

2. Enter the repository directory (denote as `$HOME`):

```
cd bf_workspace
```

3. Build the required Docker image in the repository root directory:

```
./build.sh
```

C. Experiment Workflow

After building the Docker image, run the basic test with the command:

```
./attack.sh none
```

Our system does not support running multiple instances (using attack.sh) in parallel, as they share the same working directory. The following outputs are expected:

```
casetype is none
[+] Running 2/2
 - Network case_default       Created     0.2s
 - Container case-ethmysql-1  Started     0.6s
run strategy none
INFO[0000] Specified a chain config file: /root/confi
g/config.yml  prefix=genesis
INFO[0000] No genesis time specified, defaulting to n
ow()  prefix=genesis
INFO[0000] Delaying genesis 1752219566 by 15 seconds
...
INFO[0000] Command completed
prefix=genesis
[+] Running 17/17
 - Network none_meta            Created   0.2s
 - Container none-execute3-1    Started   1.3s
 ...
 - Container none-validator2-1  Started   3.5s
wait 360 seconds
```

[7]Please ensure install unzip in your environment.

After running for six minutes, the experiment stops, and the output is as follows:

```
[+] Running 17/17
 - Container none-attacker1-1   Removed   0.7s
 - Container none-validator4-1  Removed  12.0s
 ...
 - Container none-execute1-1    Removed  10.9s
 - Network none_meta            Removed   0.8s
result collect
[+] Running 2/2
 - Container case-ethmysql-1  Removed     2.2s
 - Network case_default       Removed     0.9s
```

D. Major Claims

(C1): *BunnyFinder can reproduce known incentive attacks. This is proven by experiment (E1), as described in Section V in the paper.*

(C2): *BunnyFinder can discover three previously unknown incentive flaws. This is proven by experiment (E2), which reproduces the results in Section V in the paper.*

(C3): *BunnyFinder generates and evaluates attack instances with high coverage. This is proven by experiment (E3), which reproduces the results in Section V in the paper.*

(C4): *BunnyFinder can improve the attack effects using reinforcement learning. This is proven by experiment (E4), which reproduces Table III in Section VI in the paper.*

E. Evaluation

1) Experiment (E1): [30 human-minutes + 3 compute-hours]: Reproduce five known incentive attacks, such as exante reorg attack, sandwich reorg attack, and staircase attack.

[Preparation] If the test experiment is successful, no additional preparation is required.

[Execution] The experiments need to start each attack manually. Each attack reproduction will run for one hour. The output should be similar to that in the basic test. Run each attack experiment under `$HOME`.

Run the exante reorg attack in Prysm 5.2.0 for one hour by:

```
./attack.sh exante
```

Run the sandwich reorg attack in Prysm 5.2.0 for one hour by:

```
./attack.sh sandwich
```

Run the staircase attack in Prysm 4.0.5 for one hour by:

```
./attack.sh staircase
```

[Results] After completion, the output will be displayed as follows:

```
[+] Running 17/17
 - Container exante-validator5-1  Removed 12.7s
 - Container exante-validator3-1  Removed 12.5s
 ...
 - Container exante-execute5-1    Removed 10.8s
 - Network exante_meta            Removed  0.8s
result collect
exante attack occurs reorganize blocks in slot 8-9.
exante attack occurs reorganize blocks in slot 23-24.
...
test finished and all nodes data in \$HOME/results/e
xante
[+] Running 2/2
 - Container case-ethmysql-1  Removed           2.0s
 - Network case_default       Removed
```

The reorganization of blocks from honest validators indicates that the attack has been successfully reproduced.

2) Experiment (E2): [30 human-minutes + 3 compute-hours]: Identify three new incentive attacks, including selfish mining attack, staircase attack-II, and pyrrhic victory attack.

[Preparation] If the test experiment and experiment E1 are successful, no additional preparation is required.

[Execution] The experiments need to start each attack manually. Each attack reproduction will run for one hour. The output should be similar to that in the basic test and experiment E1. Run each attack experiment in the `$HOME` directory.

Run the selfish mining attack in Prysm 5.2.0 for one hour by:

```
./attack.sh selfish
```

Run the staircase attack-II in Prysm 5.2.0 for one hour by:

```
./attack.sh staircase-ii
```

Run the pyrrhic victory attack in Prysm 5.2.0 for one hour by:

```
./attack.sh pyrrhic-victory
```

[Results] After completion, the output will be displayed as follows:

```
[+] Running 17/17
 - Container staircaseii-validator5-1  Removed 12.7s
 - Container staircaseii-validator3-1  Removed 12.5s
 ...
 - Container staircaseii-execute5-1    Removed 10.8s
 - Network staircaseii_meta            Removed  0.8s
result collect
staircaseii attack occurs reorganize blocks in slot
152-216.
staircaseii attack occurs reorganize blocks in slot
542-595.
test finished and all nodes data in /home/ec2-user/
bf_workspace/results/staircaseii
[+] Running 2/2
 - Container case-ethmysql-1  Removed           2.0s
 - Network case_default       Removed
```

The reorganization of blocks from honest validators indicates that the attack has been successfully conducted.

3) Experiment (E3): [30 human-minutes]: Query and analyze attack instances from our attack database. The database contains exactly 7,991 completed attack strategy records shown in the paper. One can execute SQL statements to view our attack database.

[Preparation] If the test experiment and experiments E1&E2 are successful, no additional preparation is required.

[Execution] Connect to the remote database and execute SQL statements for querying attack strategies. Run each query under `$HOME`.

Connect to the remote database by:

```
export MYSQL_PASSWORD=j8P#zQ7@mV2kL9xxD
```

Use the script to connect to our remote database:

```
./tool/connect_ndss.sh
```

Query the top 10 most effective attack strategies ordered by honest validator loss rate:

```
SELECT uuid,category,honest_lose_rate_avg,
attacker_lose_rate_avg FROM t_strategy ORDER BY
honest_lose_rate_avg DESC LIMIT 10;
```

Query detailed strategy content by a specific uuid:

```
SELECT content FROM t_strategy WHERE uuid =
'your_uuid_here';
```

[Results] After completion, the output will be displayed in Figure 21.

[Note] One can view all our attack instances by querying uuid. Each attack instance is uniquely identified by its uuid and contains metadata including attack strategies, timestamps, the loss rate of honest validators and Byzantine validators.

```
mysql> SELECT COUNT(1) FROM t_strategy where
 ↪  is_end=1;
+----------+
| COUNT(1) |
+----------+
|     7991 |
+----------+
1 row in set (0.24 sec)
mysql> SELECT
 ↪  uuid,category,honest_lose_rate_avg,attacker_lose_rate_avg
 ↪  FROM t_strategy ORDER BY honest_lose_rate_avg
 ↪  DESC LIMIT 10;
+------+------+-------+-----------+
| uuid                                 | category
 ↪  | honest_lose_rate_avg | attacker_lose_rate_avg
 ↪  |
+------+------+-------+-----------+
| be0392e4-2af5-4328-ae73-75cb940183fb |
 ↪  ext_unrealized  |                    2 |
 ↪  2 |
| 8165a654 | ext_withholding |   1.4351343907591927
 ↪  |     0.4257843676895559 |
| 0908f1a1 | ext_withholding |   1.0404429370482193
 ↪  |    0.36005933160736164 |
| 36917a9b | ext_unrealized  |   0.8446009559734197
 ↪  |    0.23046993257190004 |
| 1d80817a | ext_exante      |   0.8082500263074806
 ↪  |     0.3611931605441617 |
| 04c85187 | ext_exante      |   0.8055164361051975
 ↪  |    0.21875258015687382 |
| 51939515 | ext_withholding |   0.7923948105456591
 ↪  |    0.21484450254575513 |
| a217afb6 | ext_exante      |   0.7872249249224978
 ↪  |    0.28465754717886815 |
| 9592f814 | ext_staircase   |   0.7792295550392992
 ↪  |    0.21093986514380028 |
| 03332ad2 | ext_withholding |   0.7660159949489652
 ↪  |     0.2070317875326818 |
+------+------+-------+-----------+
10 rows in set (0.19 sec)
```

Figure 21: The output of SQL results.

4) Experiment (E4): [30 human-minutes + 2 compute-hours]: Compare extended staircase attack with RL-optimized staircase attack.

[Preparation] If the test experiment and experiment E1 are successful, no additional preparation is required.

[Execution] The experiments need to be started manually by:

```
./attack.sh rl
```

[Results] After completion, the system outputs the honest validators' loss rate, the Byzantine validators' loss rate, Byzantine validators' advantage, and the success rate, both before and after reinforcement learning optimization. Notably, due to the introduction of randomness, the exact values may differ slightly from those reported in Table III of Section VI. However, the overall trend remains consistent: reinforcement learning increases the Byzantine advantage.