Understanding Ethereum Mempool Security under Asymmetric DoS by Symbolized Stateful Fuzzing

Yibo Wang and Yuzhe Tang, *Syracuse University;* Kai Li, *San Diego State University;* Wanning Ding and Zhihua Yang, *Syracuse University*

https://www.usenix.org/conference/usenixsecurity24/presentation/wang-yibo

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Understanding Ethereum Mempool Security under Asymmetric DoS by Symbolized Stateful Fuzzing

Yibo Wang
Syracuse University
ywang349@syr.edu

Yuzhe Tang ✉*
Syracuse University
ytang100@syr.edu

Kai Li
San Diego State University
kli5@sdsu.edu

Wanning Ding
Syracuse University
wding04@syr.edu

Zhihua Yang
Syracuse University
zyang47@syr.edu

Abstract

In blockchains, mempool controls transaction flow before consensus, denial of whose service hurts the health and security of blockchain networks. This paper presents MPFUZZ, the first mempool fuzzer to find asymmetric DoS bugs by exploring the space of symbolized mempool states and optimistically estimating the promisingness of an intermediate state in reaching bug oracles. Compared to the baseline blockchain fuzzers, MPFUZZ achieves a $> 100\times$ speedup in finding known DETER exploits. Running MPFUZZ on major Ethereum clients leads to discovering new mempool vulnerabilities, which exhibit a wide variety of sophisticated patterns, including stealthy mempool eviction and mempool locking. Rule-based mitigation schemes are proposed against all newly discovered vulnerabilities.

1 Introduction

In Ethereum, a mempool buffers unconfirmed transactions from web3 users before they are included in the next blocks. Mempool provides the essential functionality to bridge the gap between varying rates of submitted transactions and rates of produced blocks, regardless of public or private transactions it serves. As shown in recent studies [28], denying a mempool service can force the blockchain to produce blocks of low or even zero (Gas) utilization, undermining validators' incentives and shrinking the blockchain networks in the long run, re-introducing the 51% attacks. Besides, a denied mempool service can prevent normal transactions from block inclusion, cutting millions of web3 users off the blockchain and failing the DApps relying on real-time blockchain access.

Problem: Spamming the mempool to deny its service has been studied for long [15, 18, 23, 33]. Early designs by sending spam transactions at high prices burden attackers with high costs and are of limited practicality. What poses a real threat is Asymmetric DeniAl of Mempool Service, coined by ADAMS, in which the mempool service is denied at an asymmetrically low cost. That is, the attack costs, in terms of the fees of adversarial transactions, are significantly lower than those of normal transactions victimized by the denied mempool. In the existing literature, DETER [28] is the first ADAMS attack, and it works by sending invalid transactions to *directly evict* normal transactions in the mempool. MemPurge [35] is a similar mempool attack that finds a way to send overdraft transactions into Geth's pending transaction pool and causes eviction there. These known attacks are easy to detect (i.e., following the same direct-eviction pattern). In fact, the DETER bugs reported in 2021 have been successfully fixed in all major Ethereum clients as of Fall 2023, including Geth, Nethermind, Erigon, and Besu. Given this state of affairs, we pose the following research question: Are there new ADAMS vulnerabilities in the latest Ethereum clients already patched against direct-eviction based attacks?

This work takes a systematic and semi-automated approach to discovering ADAMS vulnerabilities, unlike the existing DETER bugs that are manually found. Fuzzing mempool implementations is a promising approach but also poses unique challenges: Unlike the consensus implementation that reads only valid confirmed transactions, the mempool, which resides in the pre-consensus phase, needs to handle various unconfirmed transactions, imposing a much larger input space for the fuzzer. For instance, a mempool can receive *invalid transactions under legitimate causes*,[1] and factors such as fees or prices are key in determining transaction admission outcomes. Existing blockchain fuzzers including Fluffy [36], Loki [29] and Tyr [25] all focus on fuzzing consensus implementation and don't explore the extra transaction space required by mempool fuzzing. As a result, directly re-purposing a consensus fuzzer to fuzz mempool would be unable to detect the DETER bugs as evaluated in Appendix § A, let alone discover more sophisticated new ADAMS bugs.

Proposed methods: To efficiently fuzz mempools, our key observation is that real-world mempool implementation admits transactions based on abstract "symbols", such as pending

*✉ Yuzhe Tang is the corresponding author.

[1] For instance, future transactions can be caused by out-of-order information propagation in Ethereum.

and future transactions, instead of concrete value. Thus, *sending multiple transactions under one symbol would trigger the same mempool behavior repeatedly*, and it suffices to explore just one transaction per symbol during fuzzing without losing the diversity of mempool behavior.

We propose symbolized-stateful mempool fuzzing or MPFUZZ. To begin with, MPFUZZ is set up and run in a three-step workflow: It is first manually set up against a size-reduced mempool under test, which induces a much smaller search space for fuzzing. Second, MPFUZZ is iteratively run against the reduced mempool to discover short exploits. Third, the short exploits are manually extended to actual ones that are functional on real mempools and Ethereum clients. Internally, the design of MPFUZZ is based on seven transaction symbols we design out of Ethereum semantics: future, parent, overdraft, latent overdraft, and replacement transactions. Under these symbols, in each iteration of fuzzing, MPFUZZ explores one concrete transaction per symbol, sends the generated transaction sequence to the target mempool, observes the mempool end state, and extracts the feedback of symbolized state coverage and state promisingness in reaching bug oracles to guide the next round of fuzzing. In particular, MPFUZZ employs a novel technique to evaluate state promisingness, that is, by *optimistically* estimating the costs of unconfirmed transactions whose validity is subject to change in the future.

Found attacks: We run MPFUZZ on six leading execution-layer Ethereum clients deployed on the mainnet's public-transaction path (Geth [7], Nethermind [11], Erigon [5], Besu [10], Reth [13], and OpenEthereum [12]), three PBS builders (proposer-builder separation) on the mainnet's private-transaction and bundle path (Flashbot $v1.11.5$ [6], EigenPhi [4] and bloXroute [1]), and three Ethereum-like clients (BSC $v1.3.8$ [2] deployed on Binance Smart Chain, go-opera $v1.1.3$ [9] on Fantom, and core-geth $v1.12.19$ [3] on Ethereum Classic). Compared to the baseline fuzzers, MPFUZZ can find exploits faster by more than two orders of magnitude. On Ethereum clients of historical versions, MPFUZZ can rediscover known DETER bugs. On the latest Ethereum clients where DETER bugs are fixed, MPFUZZ can find new ADAMS attacks, described next.

We summarize the newly discovered ADAMS exploits into several patterns: 1) Indirect eviction by valid-turned-invalid transactions. Unlike the direct-eviction pattern used in DETER, the indirect-eviction attack works more stealthily in two steps: The attacker first sends normal-looking transactions to evict victim transactions from the mempool and then sends another set of transactions to *turn* the admitted normal-looking transactions into invalid ones, bringing down the cost. 2) Locking mempool, which is to occupy the mempool to decline subsequent victim transactions. Mempool locking does not need to evict existing transactions from the mempool as the eviction-based DETER attacks do. 3) Adaptive attack strategies where the attack composes adversarial transactions in an adaptive way to the specific policies and implementations of the mempool under test. Example strategies include composing transactions of multiple patterns in one attack, resending evicted transactions to evict "reversible" mempools, locking mempool patched against turning, etc. We propose rule-based mitigation schemes against all newly discovered vulnerabilities.

All newly found ADAMS bugs are reported to the developer communities, including Ethereum Foundation, BSC, Fantom, PBS builders, etc. with 15 bugs confirmed and 4 fixed in recent client releases [8]. Bug reporting is documented on the webpage [20], which also includes demos of MPFUZZ on the tested Ethereum clients. MPFUZZ will be open-sourced to facilitate vulnerability discovery in more and future clients.

Contributions: The paper makes contributions as follows:

- *New fuzzing problem*: This paper is the first to formulate the mempool-fuzzing problem to automatically find asymmetric DoS vulnerabilities as bug oracles. Fuzzing mempools poses new unique challenges that existing blockchain fuzzers don't address and entails a larger search space including invalid transactions and varying prices.

- *New fuzzing method*: The paper presents the design and implementation of MPFUZZ, a symbolized-stateful mempool fuzzer. Given a mempool implementation, MPFUZZ defines the search space by the mempool states covered under symbolized transaction sequences and efficiently searches this space using the feedback of symbolized state coverage and the promising-ness of an intermediate state in triggering bug oracles. With a Python prototype and runs on real Ethereum clients, MPFUZZ achieves a $>100\times$ speedup in finding known DETER exploits compared to baselines.

- *New discovery of mempool vulnerabilities*: MPFUZZ discovers new asymmetric-DoS vulnerabilities in six major Ethereum clients in the mainnet. By evaluation under real transaction workloads, all found attacks achieve $84.6-99.6\%$ success rates and low costs such as adversarial transaction fees $100\times$ lower than the victim transaction fees.

2 Related Works

Consensus fuzzers: Fluffy [36] is a code-coverage based differential fuzzer to find consensus bugs in Ethereum Virtual Machines (EVM). Specifically, given an EVM input, that is, a transaction sequence, Fluffy sends it to multiple nodes running different Ethereum clients (e.g., Geth and OpenEthereum) for execution. 1) The test oracle in Fluffy is whether the EVM end states across these nodes are different, implying consensus failure. 2) Fluffy mutates ordered transaction sequences at two levels: it reorders/adds/deletes transactions in the sequence, and for each new transaction to generate, it randomly selects values in Gas limits and Ether amounts. For the data field, it employs semantic-aware strategies to add/delete/mutate bytecode instructions of pre-fixed smart contract templates. 3) Fluffy uses code coverage as feedback to guide the mutation.

Loki [29] is a stateful fuzzer to find consensus bugs causing crashes in blockchain network stacks. Specifically, Loki runs as a fuzzer node interacting with a tested blockchain node through sending and receiving network messages. 1) The test oracle in Loki is whether the tested blockchain node crashes. 2) The fuzzer node uses a learned model of blockchain network protocol to generate the next message of complying format, in which the content is mutated by randomly choosing integers and bit-flipping strings. That is, the mutation in Loki is unaware of application level semantics. For instance, if the message is about propagating an Ethereum transaction, the transaction's nonce, Ether amount, and *GasPrice* are all randomly chosen; the sender, receiver, and data are bit flipped. 3) Loki records messages sent and received from the test node. It uses them as the state. New states receive positive feedback.

Tyr [25] is a property-based stateful fuzzer that finds consensus bugs causing the violation of safety, liveness, integrity, and other properties in blockchain network stacks. Specifically, in Tyr, a fuzzer node is connected to and executes a consensus protocol with multiple neighbor blockchain nodes. 1) The test oracle Tyr uses is the violation of consensus properties, including safety (e.g., invalid transactions cannot be confirmed), liveness (e.g., all valid transactions will be confirmed), integrity, etc. The checking is realized by matching an observed end state with a "correct" state defined and run by the Tyr fuzzer. 2) Tyr mutates transactions on generic attributes like randomly selecting senders and varying the amount of cryptocurrencies transferred (specifically, with two values, the full sender balance, and balance plus one). When applied to Ethereum, Tyr does not mutate Ethereum-specific attributes, including nonce or *GasPrice*. It generates one type of invalid transaction, that is, the double-spending one, which in account-based blockchains like Ethereum are replacement transactions of the same nonce. 3) Tyr uses state divergence as the feedback, that is, the difference of end states on neighbor nodes receiving consensus messages in the same iteration. In Tyr, the states on a node include both the messages the node sent and received, as well as blocks, confirmed transactions, and other state information.

These existing blockchain fuzzers [25,29,36] cannot detect mempool DoS bugs and are different from this work. Briefly, mempool DoS does not trigger system crashes and can not be detected by Loki. Mempool content difference across nodes does not mean insecurity, and the mempool DoS bugs can not be detected by Fluffy (which detects consensus state difference). While Tyr aims to detect liveness and safety violations, their model of invalid transactions is rudimentary. Specifically, they only model double-spending transactions and cannot detect DETER bugs [28] that rely on more advanced invalid transactions, like future transactions and latent overdrafts.

Blockchain DoS: Blockchain DoS security has been examined at different system layers, including P2P networks [21, 26, 30, 34], mining-based consensus [14, 31], and application-level smart contracts [17, 24, 32] and DApp (decentralized application) services [27]. These DoSes are not related to mempools and are orthogonal to this work.

For mempool DoS, a basic attack is by sending spam transactions with high fees to evict victim transactions of normal fees [15, 18, 23, 33], which incur high costs to attackers. DETER [28] is the first asymmetric DoS on Ethereum mempool where the adversary sends invalid transactions of high fees (e.g., future transactions or latent overdrafts) to evict victim transactions of normal fees. Concurrent to this work is MemPurge [35] posted online in June 2023; in it, the attacker reconnects her adversarial future transactions and makes them latent-overdraft transactions. The DETER bugs have been fixed on the latest Ethereum clients such as Geth $v1.11.4$, as we tested them in July 2023. Both DETER and MemPurge are manually discovered.

3 Background

Ethereum mempools: In Ethereum, users send their transactions to the blockchain network, which are propagated to reach validator nodes. On every blockchain node, unconfirmed transactions are buffered in the mempool before they are included in the next block or evicted by another transaction. In Ethereum 2.0, transactions are propagated in two fashions: public transactions are broadcast among all nodes, and private transactions are forwarded to selected nodes, more specifically, selected builders and proposers (as in PBS or proposer-builder separation). The mempool is present on both paths, and it is inside Ethereum clients such as Geth [7] and Nethermind [11] (handling public transactions), and Flashbot [6] (private transactions). On these clients, a mempool also serves many other downstream modules, including MEV searchers or bots (in PBS), Gas stations, RPC queries, etc.

Mempools on different nodes are operated independently and don't need to synchronize as the consensus layer does. For instance, future transactions are not propagated, and the set of future transactions in the mempool on one node is different from another node.

The core mempool design is transaction admission: Given an initial state, whether and how the mempool admits an arriving transaction. In practice, example policies include those favoring admitting transactions of higher prices (and thus evicting or declining the ones of lower prices), transactions arriving earlier, certain transaction types (e.g., parent over child transactions), and valid transactions (over future transactions). See different types of Ethereum transactions in § 4.1.

Transactions and fees: In Ethereum, Gas measures the amount of computations caused by smart contract execution. When preparing for transaction *tx*, the sender needs to specify *Gas* and *GasPrice*; the former is the maximal amount of computations allowed for running smart contracts under *tx*, and the latter is the amount of Ether per Gas the sender is willing to pay. After *tx* is included in a block, the actual amount of computations consumed by contract execution is denoted by $GasUsed$. The fees of transaction tx are the product of $GasUsed$ and $GasPrice$, that is, $tx.fee = GasUsed * GasPrice$.

After EIP-1559 [19], $GasPrice$ is divided into two components: $BasePrice$ and $PriorityPrice$, that is, $GasPrice = BasePrice + PriorityPrice$. Blockchain nodes follow the Ethereum protocol to derive $BasePrice$ from Gas utilization in recent blocks. The fees associated with the $BasePrice$ are burnt upon transaction inclusion. $PriorityPrice$ is set by the transaction sender.

4 Threat Model and Bug Oracle

In the threat model, an attacker controls one or few nodes to join an Ethereum network, discovers and neighbors critical nodes (e.g., top validators, backends of an RPC Service, MEV searcher) if necessary, and sends crafted adversarial transactions to their neighbors. The attacker in this work has the same networking capacity as in DETER [28].

The attacker's goal is to disable the mempools on the critical or all nodes in an Etheurem network and cause damage such as dropped transactions in block inclusion, produced blocks of low or zero (Gas) utilization, and disabled other downstream services. In practice, this damage is beneficial to competing block validators, MEV searchers, and RPC services. Besides, the attacker aims to cause this damage at asymmetrically low costs, as will be described.

4.1 Notations

Transactions: In Ethereum, we consider two types of accounts: a benign account of index i denoted by B_i, and an adversarial account of index i denoted by A_i. A (concrete) Ethereum transaction tx is denoted by $tx\left[\begin{smallmatrix} s & v \\ n & f \end{smallmatrix}\right]$, where the sender is s, the nonce is n, the $GasPrice$ is f, and the transferred value in Ether is v. This work does not consider the "data" field in Ethereum transactions or smart contracts because smart contracts do not affect the validity of the Ethereum transaction. The notations are summarized in Table 1.

Table 1: Notations

Symbol	Meaning	Symbol	Meaning
m	Mempool length	ops	Tx sequence
st	State of residential txs in mempool	dc	History of declined txs from mempool
A	Adversarial tx sender	B	Benign tx sender
$tx\left[\begin{smallmatrix} s & v \\ n & f \end{smallmatrix}\right]$	A transaction from sender s, of nonce n, transferring Ether value v, and with $GasPrice$ f		

We use each of the following symbols to represent a disjoint group of transactions. Informally, $\mathcal{N}$ is a transaction sent from a benign account, $\mathcal{F}$ a future transaction, $\mathcal{P}$ a parent transaction, $\mathcal{C}$ a child transaction, $\mathcal{O}$ an overdraft transaction, $\mathcal{L}$ a latent overdraft transaction, and $\mathcal{R}$ a replacement transaction. These symbols are formally specified in § 5.

Particularly, we use the notion of latent overdraft from the existing work [28], which indicates a child transaction that by itself does not overdraft but does overdraft when taking into account its parent transactions. Suppose Alice has a balance of 5 Ether and sends tx_1 of nonce 1 spending 2 Ether and tx_2 of nonce 2 spending 4 Ether. tx_2 is a latent overdraft.

States and transitions: We recognize two mempool-related states in Ethereum, the collection of transactions stored in mempool st, and the collection of declined transactions dc.

Suppose a mempool of state $\langle st_i, dc_i \rangle$ receives an arriving transaction tx_i and transitions to state $\langle st_{i+1}, dc_{i+1} \rangle$.

An arriving transaction can be admitted with eviction, admitted without eviction, or declined by a mempool. The three operations are defined as follows. 1) An arriving transaction tx_i is admitted with evicting a transaction tx_i' from the mempool, if $st_{i+1} = st_i \setminus tx_i' \cup tx_i, dc_{i+1} = dc_i$. 2) tx_i is admitted without evicting any transaction in the mempool, if $st_{i+1} = st_i \cup tx_i, dc_{i+1} = dc_i$. 3) tx_i is declined and does not enter the mempool, if $st_{i+1} = st_i, dc_{i+1} = dc_i \cup tx_i$.

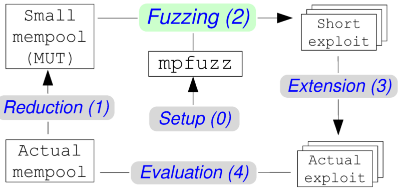


Figure 1: Overview of exploit discovery and evaluation workflow: 0) MPFUZZ setup, 1) mempool reduction, 2) fuzzing on reduced mempool under test (MUT) to discover short exploits, and 3) exploit extension. The extended exploits are 4) evaluated on the actual mempool of the original size. Green means automated tasks, and gray requires manual effort.

4.2 Exploit Discovery Workflow

To set up the stage, we present the workflow overview in this work. The workflow is depicted in Figure 1. To discover vulnerabilities and exploits in an actual mempool, one has to 0) set up MPFUZZ by various parameters (as will be introduced), 1) reduce the mempool from its original configuration to a much smaller version, named mempool under test (MUT). 2) MPFUZZ is run on the reduced MUT deployed locally (with settings described in § 7.1). Fuzzing leads to the discovery of short exploits. 3) Short exploits are extended to the longer ones, that is, actual exploits. 4) The success of the actual exploit is evaluated on the actual mempools deployed in close-to-operational settings, such as using popular Ethereum testnets like Goerli (see § 6.3.1) or a network we set up locally running unmodified Ethereum clients (see § 6.4.2 and § D). Fuzzing a reduced mempool (MUT) instead of an original one is necessary to ensure the efficient execution of the MPFUZZ, which entails many iterations of re-initiating and populating the mempool. We discuss and evaluate the possible false positives that could be introduced by fuzzing smaller MUT in

§ 7.2.

In this workflow, fully automated is Step 2), that is, the discovery of short exploits on a given MUT (colored green in Figure 1). Other steps colored in gray require manual efforts and are described below: In **Step 0)**, MPFUZZ setup entails setting parameters in bug oracles (e.g., ε and λ) and fixed symbols in guiding fuzzing. The setup is usually one-time. In **Step 1)**, mempool reduction entails reconfiguring the mempool to a much smaller capacity (e.g., MUT length $m = 6$ or $m = 16$, compared to the original size like $m' = 6144$ as in Geth). In addition, policy-specific parameters in the actual mempool are tuned down proportionally. For instance, the Geth mempool buffers up to $py'_1 = 1024$ future transactions, and when buffering more than $py'_3 = 5120$ pending transactions, the mempool triggers a protective policy to limit the number of transactions sent from the same sender by $py'_2 = 16$. In a reduced MUT of $m = 6$, these parameters are reduced while retaining the same proportion, such as $py_1 = 1, py_3 = 5, py_2 = 2$. The mempool reduction requires expert understanding and manual efforts, and it is by nature the best effort. In **Step 3)**, it first de-duplicates the raw short exploits found by MPFUZZ. Then, for each distinct short exploit, it extends it by repeating the transaction-admission events to fit into the actual mempool, during which one may practice strategies like increasing nonce, switching sender accounts, and measuring normal transactions in the actual mempool. In exploit extension, one could also consider the factors not modeled in MUT or MPFUZZ such as *BasePrice*, as seen in the case in § 6.4.

4.3 Bug Oracle: Definitions & Rationale

Definition 4.1 (Tx admission timeline) *In a transaction admission timeline, a mempool under test is initialized at state $\langle st_0, dc_0 = \varnothing\rangle$, receives a sequence of arriving transactions ops, and ends up with state $\langle st_n, dc_n\rangle$. Then, a validator continually builds blocks by selecting and clearing transactions in the mempool until the mempool is empty. This transaction timeline is denoted by $\langle st_0, dc_0 = \varnothing\rangle, ops \Rightarrow \langle st_n, dc_n\rangle$.*

The transaction admission timeline is simplified from the timeline that an operational mempool experiences where transaction arrival and block building can be interleaved. The simplification is intended to trade off the accuracy of fuzzing results for fuzzing efficiency as discussed in § 7.2.

We use two mempool-attack templates to define two bug oracles: an eviction-based DoS where adversarial transactions evict existing victim transactions in the mempool, and a locking-based DoS where existing adversarial transactions in the mempool decline arriving victim transactions.

Definition 4.2 (Eviction bug oracle) *A transaction admission timeline, $\langle st_0, dc_0 = \varnothing\rangle, ops \Rightarrow \langle st_n, dc_n\rangle$, is a successful* ADAMS *eviction attack, iff. 1) the admission timeline causes full damage, that is, initial state st_0 contains only benign transactions, ops are adversarial transactions, and the result end state st_n contains only adversarial transactions (in Equation 1), and 2) the attack cost is asymmetrically low, that is, the total adversarial transaction fees in the end state st_n to be charged are smaller, by a multiplicative factor ε, than the attack damage measured by the fees of evicted transactions in the initial state st_0 (in Equations 2 and 3). Formally,*

$$
\begin{aligned}
st_0 \cap st_n &= \varnothing && (1)\\
asym_E(st_0, ops) &\overset{def}{=} \frac{\sum_{tx \in st_n} tx.fee}{\sum_{tx \in st_0} tx.fee} && (2)\\
asym_E(st_0, ops) &< \varepsilon && (3)
\end{aligned}
$$

Definition 4.3 (Locking bug oracle) *A transaction admission timeline, $\langle st_0 = \varnothing, dc_0 = \varnothing\rangle, ops \Rightarrow \langle st_n, dc_n\rangle$, is a successful* ADAMS *locking attack, iff. 1) the admission timeline causes full damage, that is, the mempool is first occupied by adversarial transactions (i.e., st_n contains only adversarial transactions) and then declines the arriving normal transactions (i.e., dc_n contains only normal transactions); the normal transactions and adversarial transactions are sent by different accounts (i.e., Equation 4), and 2) the attack cost is asymmetrically low, that is, the average adversarial transaction fees in the mempool are smaller, by a multiplicative factor λ, than the attack damage measured by the average fees of victim transactions declined (i.e., Equations 5 and 6). Formally,*

$$
\begin{aligned}
\cup_{tx \in dc_n} tx.sender &\cap \cup_{tx \in st_n} tx.sender = \varnothing && (4)\\
asym_D(ops) &\overset{def}{=} \frac{\sum_{tx \in st_n} tx.fee/\|st_n\|}{\sum_{tx \in dc_n} tx.fee/\|dc_n\|} && (5)\\
asym_D(ops) &< \lambda && (6)
\end{aligned}
$$

Design rationale: Both definitions use the normal transaction fees to measure the attack damage because these fees are not collected by the validator in the timeline under attack and are collectible without attack. Specifically, the damage in Definition 4.2 is measured by the fees of normal transactions in st_0 which are evicted in end state st_n and which would not have been evicted had there been no adversarial transactions in *ops*. The damage in Definition 4.3 is measured by the fees of normal transactions in *ops* which are declined from end state st_n and which would not have been declined had there been no adversarial transactions in *ops*.

Both definitions are strict and require causing full damage, such as evicting or declining *all* normal transactions. Strict definitions are necessary to ensure that what is found as exploits on the small MUT is true positive and works on the actual mempool.

The targeted attacks by this work are denial of mempool service in feeding downstream validators with valid transactions sent from benign users. They are different from other forms of DoS in blockchains, including resource exhaustion via executing malicious smart contracts [27, 35].

Particularly, in the threat model of mempool DoS, we assume the validators are benign and functional in building blocks. Because in all Ethereum clients we know, transactions are admitted into mempool without execution (i.e., executing the smart contracts they invoke), *GasUsed* is not a factor in transaction admission.[2] Thus, our bug oracle does not vary *GasUsed*. Instead, all the transactions are fixed at 21000 Gas (i.e., as if they don't invoke smart contracts).[3] This design significantly reduces the search space without loss in covering different mempool behaviors. Besides, our bug oracle does not model the effect of EIP1559 or *BasePrice*. Without modeling *BasePrice*, our bug oracle can still capture the alternative transaction fee (the product of *GasPrice* and 21000 Gas), which is still proportional to actual *PriorityPrice* or damage on validator revenue.

5 Stateful Mempool Fuzzing by MPFUZZ

5.1 Transaction Symbolization

The key challenge in designing MPFUZZ is the large input space of transaction sequences. Recall that an Ethereum transaction consists of multiple attributes (sender, nonce, *GasPrice*, value, etc.), each defined in a large domain (e.g., 64-bit string). Exploring the raw transaction space is hard; the comprehensive search is inefficient, and randomly trying transactions as done in the state-of-the-art blockchain fuzzers (in § 2) is ineffective.

Intuition: We propose symbolizing transactions for efficient and effective mempool stateful fuzzing. Our key idea is to map each group of concrete transactions, triggering an equivalent mempool behavior into a distinct symbol so that searching for one transaction is sufficient to cover all other transactions under the same symbol. Transaction symbolization is expected to reduce the search space from possible concrete transactions to the symbol space.

In this work, we manually design seven symbols to represent transactions based on the Ethereum "semantics", that is, how different transactions are admitted by the mempool.

Specification: Suppose the current state contains adversarial transactions sent from r accounts $A_1, \dots, A_r$. These accounts have an initial balance of m Ether, where m equals the mempool capacity (say $m = 1000$). The rationale is that these accounts can send at most m adversarial transactions, each minimally spending one Ether, to just occupy a mempool of m slots. A larger value of attacker balance is possible but increases the search space.

Symbol $\mathcal{N}$ defines a transaction subspace covering any normal transactions sent from any benign account and of any nonce, namely $tx\begin{bmatrix} B* & * \\ * & * \end{bmatrix}$. In MPFUZZ, Symbol $\mathcal{N}$ is instantiated to concrete transactions of fixed *GasPrice* 3 wei and of value 1 Ether. That is, symbol $\mathcal{N}$ is instantiated to transaction $tx\begin{bmatrix} B* & 1 \\ * & 3 \end{bmatrix}$, as shown in Table 2. Among all symbols, $\mathcal{N}$ is the only symbol associated with benign sender accounts.

Table 2: Symbols, transactions, and associated costs. Tx refers to the instantiated transaction under a given symbol.

Symbol	Description	Tx	*cost*()	*opcost*()
$\mathcal{N}$	Benign	$tx\begin{bmatrix} B* & 1 \\ * & 3 \end{bmatrix}$	3	3
$\mathcal{F}$	Future	$tx\begin{bmatrix} A* & 1 \\ m+1 & m+4 \end{bmatrix}$	0	0
$\mathcal{P}$	Parent	$tx\begin{bmatrix} A>r & 1 \\ 1 & [4,m+3] \end{bmatrix}$	$[4, m+3]$	$[4, m+3]$
$\mathcal{C}$	Child	$tx\begin{bmatrix} A[1,r] & 1 \\ \geq 2 & m+4 \end{bmatrix}$	$m+4$	1
$\mathcal{O}$	Overdraft	$tx\begin{bmatrix} A[1,r] & m+1 \\ \geq 2 & m+4 \end{bmatrix}$	0	0
$\mathcal{L}$	Latent overdraft	$tx\begin{bmatrix} A[1,r] & m-1 \\ \geq 2 & m+4 \end{bmatrix}$	0	0
$\mathcal{R}$	Replacement	$tx\begin{bmatrix} A[1,r] & m-1 \\ 1 & m+4 \end{bmatrix}$	$m+4$	0

Symbol $\mathcal{F}$ defines the transaction subspace of any future transaction sent from an adversarial account; the future transaction is defined w.r.t. the current state. MPFUZZ instantiates symbol $\mathcal{F}$ to concrete transactions of nonce $m+1$, value 1 Ether, *GasPrice* $m+4$ wei, and any adversarial account $A*$. The instantiating pattern for symbol $\mathcal{F}$ is $tx\begin{bmatrix} A* & 1 \\ m+1 & m+4 \end{bmatrix}$.

Symbol $\mathcal{P}$ defines the transaction subspace of any parent transaction from an adversarial account w.r.t. the current state. MPFUZZ instantiates $\mathcal{P}$ to a transaction of Pattern $tx\begin{bmatrix} A\geq r & 1 \\ 1 & [4,m+3] \end{bmatrix}$. Here, the transaction is fixed with a *GasPrice* in the range $[4, m+3]$ wei, value at 1 Ether, and nonce at 1. The *GasPrice* of a future transaction ($\mathcal{F}$) is $m+4$ wei, which is higher than the price of a parent transaction ($\mathcal{P}$) in $[4, m+3]$ wei. The purpose is to ensure that a future transaction can evict any parent transaction during fuzzing.

Symbol $\mathcal{C}$ defines the transaction subspace of any adversarial child transaction w.r.t. the state. MPFUZZ instantiates $\mathcal{C}$ to a transaction of Pattern $tx\begin{bmatrix} A[1,r] & 1 \\ \geq 2 & m+4 \end{bmatrix}$. The instantiated child transaction is fixed at *GasPrice* of $m+4$

Symbol $\mathcal{O}$ defines the transaction subspace of any adversarial overdraft transaction w.r.t. state st. MPFUZZ instantiates $\mathcal{O}$ to a transaction of Pattern $tx\begin{bmatrix} A[1,r] & m+1 \\ \geq 2 & m+4 \end{bmatrix}$. Recall that all adversarial accounts have an initial balance of m Ether.

Symbol $\mathcal{L}$ defines the transaction subspace of any adversarial latent-overdraft transaction w.r.t. state st. MPFUZZ instantiates $\mathcal{L}$ to a transaction of Pattern $tx\begin{bmatrix} A[1,r] & m-1 \\ \geq 2 & m+4 \end{bmatrix}$. All adversarial accounts have an initial balance of m Ether.

Symbol $\mathcal{R}$ defines the transaction subspace of any adversarial "replacement" transaction w.r.t., the state. Unlike the symbols above, the transactions under Symbol $\mathcal{R}$ must share the same sender and nonce with an existing transaction in the current state st. MPFUZZ instantiates $\mathcal{R}$ to a transaction of Pattern $tx\begin{bmatrix} A[1,r] & m-1 \\ 1 & m+4 \end{bmatrix}$. Here, we simplify the problem and only consider replacing the transaction with nonce 1.

Given that a concrete transaction in this work consists of four attributes, transactions are defined in a four-dimensional

[2] The further design rationale for admitting transactions without execution is that an Ethereum transaction tx's *GasUsed* is non-deterministic unless the ordering of tx w.r.t. other transactions in the block is fixed and tx is executed.

[3] We may use price and fees interchangeably.

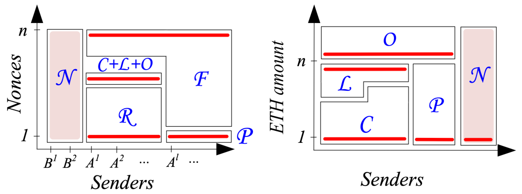


Figure 2: Symbols and transaction space reduction.

space. We visualize transaction symbols (with an incomplete view) in two two-dimensional spaces in Figure 2, that is, one of transaction sender and nonce and the other of sender and value. For each symbol, the figures depict the transaction subspace covered by the symbol (in white shapes of black lines) and the transaction pattern instantiated by MPFUZZ under the symbol (in red lines or shapes). In our design, the instantiated transactions are a much *smaller subset* of the defining transaction space. For instance, given state *st* of transactions sent from account $A_1, \ldots, A_r$, transaction $tx\begin{bmatrix} A_{r+1} & * \\ 3 & * \end{bmatrix}$ is within the defining space of symbol $\mathcal{F}$, but MPFUZZ does not instantiate $\mathcal{F}$ by transaction $tx\begin{bmatrix} A_{r+1} & * \\ 3 & * \end{bmatrix}$ (but instead to transactions $tx\begin{bmatrix} A* & 1 \\ m+1 & m+4 \end{bmatrix}$). The design rationale of transaction symbolization is deferred to Appendix A.1.

5.1.1 State Search Algorithm

Algorithm 1 MPFUZZ(SeedCorpus *sdb*, Mempool *mp*)

```
 1: sdb.init(mp);
 2: while !sdb.is_empty() or timeout do
 3:     st, ops = sdb.next();                    ▷ Selection by energy
 4:     for all ops′, st′ = mutateExec(ops, st) do
 5:         if is_ADAMS(st′) then                 ▷ Test oracle
 6:             emit(ops′, st′, "Found an exploit");
 7:         else
 8:             if !sdb.feedback(st′, sdb) then
 9:                 sdb.add(st′, ops′);
10:             end if
11:         end if
12:     end for
13: end while
```

Algorithm overview: We propose a stateful fuzzing algorithm, listed in Algorithm 1, to selectively explore the input space in a way that prioritizes new and promising mempool states towards triggering ADAMS oracle.

The core data structure is a seed corpus or *sdb* which stores a list of input-state pairs or seeds. Upon running the algorithm, the corpus maintains the mempool states covered so far by the algorithm execution. The algorithm runs an outer loop that continues until the corpus is empty, or only the states of zero energy are left, or timeout. In each iteration, the algorithm retrieves the next seed from the corpus based on how promising the seed is in reaching a state triggering the oracle (a.k.a., the energy [22]). A seed consists of a symbolized input *ops*, which consists of transaction symbols, and a symbolized state *st* reached by running a transaction sequence instantiated from the input *ops* (Line 3). The algorithm runs an inner loop, in each iteration of which it mutates the input in the current seed ($\langle st, in \rangle$) and executes the mutated input against a reinitialized empty mempool, producing end state st' (Line 4). If the end state satisfies ADAMS conditions (i.e., the test oracle), it emits the mutated input-state pair ops', st' as a newly found exploit. The algorithm further checks the feedback: The feedback is positive if the mutated input ops' brings the mempool state st' closer to triggering the test oracle than st. This entails mutated state st' to be different from state st (i.e., increased state coverage) and state st' to achieve larger damage or lower attack cost. In case of positive feedback, the algorithm would add mutated input-state pair ops', st' to the corpus.

This algorithm assumes the mempool is deterministic. That is, given a seed $\langle ops, st \rangle$, running the same input *ops* against an empty mempool multiple times always results in the same end state *st*. In practice, we generate inputs to avoid the non-deterministic behavior of real mempool implementations.

The algorithm can be configured with initial seeds and input-mutation strategies. By default, we use one initial seed whose input fills up the mempool with normal transactions. The default input-mutation strategy is to append the current input with a newly generated transaction.

Algorithm 2 mutateExec(SymbolInput *ops*, SymbolState *st*)

```
1: ops′=mutateSymbol(ops);
2: st_c=executeUnappended(ops, ops′);
3: st′_c=executeAppended(ops′, st_c, st);
4: return st′_c
```

Input mutation: Given a symbolized input *ops* and symbolized state *st*, the mutation algorithm is to explore each "slightly" different input ops' and its associated state st', such that the next stage can find the inputs producing positive feedback.

Internally, the algorithm proceeds at two levels, that is, symbols and concrete value. Specifically, as shown in Algorithm 2, 1) it appends the symbolized input *ops* with a previously untried symbol, generating the "mutated" symbolized input ops' (Line 1 in Algorithm 2). In this step, the algorithm tries different symbols in the following order: $\mathcal{P}, \mathcal{L}, \mathcal{C}$. 2) It instantiates symbolized ops' to a transaction sequence and executes the sequence in the tested mempool to obtain the concrete end state (Line 2 in Algorithm 2). This step is necessary for instantiating and executing the mutation in the next step. Specifically, in this step, the algorithm instantiates symbolized input *ops* to a concrete input in_c, which is a sequence of transactions. It then drives the transactions to a reinitialized mempool for execution. It returns the resulting concrete state st_c. 3) The algorithm instantiates the appended symbols under the context of the previous state *st* and st_c. At last, it executes the appended transaction on mempool st_c to obtain the concrete end state st'_c under the mutated input (Line 3 in Algorithm 2).

Mutation feedback: We describe how mempool states are symbolized before presenting state-based feedback. In MP-

FUZZ, a symbolized state st is a list of transaction symbols, ordered first partially as follows: $\mathcal{N} \preceq \mathcal{E} \preceq \mathcal{F} \preceq \{\mathcal{P}, \mathcal{L}, \mathcal{C}$. $\mathcal{E}$ refers to an empty slot. State slots of the same symbols are independently instantiated into transactions, except for $\mathcal{P}, \mathcal{C}, \mathcal{L}$. Child transactions (i.e., $\mathcal{C}, \mathcal{L}$) are appended to their parent transaction of the same sender (i.e., $\mathcal{P}$). Across different senders, symbols are ordered by the parent's $GasPrice$. For instance, suppose a mempool stores four concrete transactions, $tx\begin{bmatrix} A_1 & * \\ 1 & 4 \end{bmatrix}, tx\begin{bmatrix} B & * \\ 100 & 100 \end{bmatrix}, tx\begin{bmatrix} A_2 & * \\ 1 & 5 \end{bmatrix}, tx\begin{bmatrix} A_1 & * \\ 2 & 10001 \end{bmatrix}$, they are mapped to four symbols, $\mathcal{P}, \mathcal{F}, \mathcal{P}, \mathcal{C}$, and are further ordered in a symbolized state by $\mathcal{F}\mathcal{P}\mathcal{C}\mathcal{P}$.

In MPFUZZ, the feedback of an input ops' is based on the symbolized end state st'. Positive feedback on state st' is determined conjunctively by two metrics: 1) State coverage that indicates state st' is not covered in the corpus, and 2) state promising-ness that indicates how promising the current state is to reach a state satisfying bug oracles.

$$\begin{aligned} feedback(st', st, sdb) \quad = \quad & st\_coverage(st', sdb) == 1 \wedge \\ & st\_promising(st', st) == 1 \end{aligned} \quad (7)$$

Specifically, state coverage is determined by straightforwardly comparing symbolized state st', as an ordered list of symbols, with all the symbolized states in the corpus. For instance, state $\mathcal{F}\mathcal{P}\mathcal{C}\mathcal{P}$ is different from $\mathcal{F}\mathcal{P}\mathcal{P}\mathcal{C}$. This also implies concrete transaction sequence $tx\begin{bmatrix} A_1 & * \\ 1 & 4 \end{bmatrix}, tx\begin{bmatrix} B & * \\ 100 & 100 \end{bmatrix}, tx\begin{bmatrix} A_2 & * \\ 1 & 5 \end{bmatrix}, tx\begin{bmatrix} A_1 & * \\ 2 & 10001 \end{bmatrix}$ is the same with $tx\begin{bmatrix} A_1 & * \\ 1 & 4 \end{bmatrix}, tx\begin{bmatrix} B & * \\ 100 & 100 \end{bmatrix}, tx\begin{bmatrix} A_2 & * \\ 1 & 6 \end{bmatrix}, tx\begin{bmatrix} A_1 & * \\ 2 & 10 \end{bmatrix}$, as they are both mapped to the same symbolized state.

How promising a state st' is (i.e., $st\_promising(st')$) is determined as follows: A mutated state st' is more promising than an unmutated state st if any one of the three conditions is met: 1) State st' stores fewer normal transactions (under symbol $\mathcal{N}$) than state st, implying more transactions evicted and higher damage (i.e., $evict\_normal(st', st) = 1$). 2) State st' declines (speculatively) more incoming normal transactions than state st, also implying higher damage (i.e., $decline\_normal(st', st) = 1$). 3) The total fees of adversarial transactions in state st' are lower than those in state st, implying lower attack costs. We consider two forms of costs: concrete cost and symbolized cost. The former, denoted by $cost(st')$, is simply the total fees of adversarial transactions instantiated from a symbolized state st'. The latter, denoted by $opcost(st')$, optimistically estimates the cost of transactions in the current state st' that contributes to a future state triggering test oracle. We will describe the symbolized cost next. Formally, state promising-ness is calculated by Equation 8.

$$\begin{aligned} st\_promising(st', st) \quad = \quad & evict\_normal(st', st) == 1 \vee \\ & decline\_normal(st', st) == 1 \vee \\ & cost(st') < cost(st) \vee \\ & opcost(st') < opcost(st) \end{aligned} \quad (8)$$

Seed energy: Recall that in MPFUZZ, the next seed is selected from the corpus based on energy. By intuition, the energy of a state is determined based on how promising the state is in triggering the test oracle. In addition to the state promising-ness used in deciding feedback, state energy incorporates fuzz runtime information, such as how many times the state has been selected.

Specifically, the energy of a seed $\langle ops, st \rangle$ is determined by Equation 9.

$$Energy(in, st) \quad = \quad b/opcost(st) \quad (9)$$

First, each seed in the corpus records how many times it has been mutated. The more symbols it has mutated in the past, the less energy the seed currently has and the lower priority it will be selected next. b can be configured differently to traverse the state tree differently. In particular, breadth-first search (BFS) is by the following configuration: $b = 1$ if at least one symbol has not been tried (for mutation) in the current seed. Otherwise, $b = 0$.

Second, we use the symbolized cost to estimate how promising a state is and use it in seed energy.

Estimate state cost: When MPFUZZ determines how promising a state is, it needs to look beyond the current state and into all possible descendant states. We propose a heuristic that optimistically estimates the descendant-state costs given a current state. The key intuition is the following: In Ethereum, the validity of a child transaction (under symbol $\mathcal{C}$) depends on its parent/ancestor transaction. Thus, even though a transaction in the current state is valid, the transaction can be "turned" into an invalid one in subsequent states. We thus attribute, optimistically, the cost of transaction $\mathcal{C}$ to value 1, so that it is preferable to a parent transaction $\mathcal{P}$ in input mutation and seed selection, and turning $\mathcal{C}$ into an invalid one like $\mathcal{L}$ or $\mathcal{F}$ is also encouraged. The cost profiles used in MPFUZZ are summarized in Table 2.

6 Found Exploits

We have run MPFUZZ across a variety of Ethereum clients, including six leading execution-layer clients on the public-transaction path of Ethereum mainnet (Geth, Besu, Nethermind, Erigon, Reth, and OpenEthereum), three PBS clients (proposer-builder separation) on the mainnet's private-transaction and bundle path (Flashbot builder $v1.11.5$ [6], EigenPhi builder [4] and bloXroute builder-ws [1]), and the clients deployed on three operational Ethereum-like networks (BSC $v1.3.8$ [2] deployed on Binance Smart Chain, go-opera $v1.1.3$ [9] on Fantom, and core-geth $v1.12.19$ [3] on Ethereum Classic).

On the six public-transaction clients, MPFUZZ leads to the discovery of 22 bugs, as listed in Table 3, including 7 conforming to the known DETER attacks on clients of historical versions and 15 new bugs on the clients of the latest versions.

Table 3: ADAMS exploit patterns found by MPFUZZ across Ethereum clients; XT_{1-7} are eviction based, and XT_{8-9} are locking based. XT_{1-3} are known patterns in DETER [28], while others are new. ✗ indicates the presence of a bug, ✓ indicates the fixing of a bug after our reporting, and ✓ indicates the fixing of a bug by previous works.

	XT_1	XT_2	XT_3	XT_4	XT_5	XT_6	XT_7	XT_8	XT_9
Geth $\geq v1.11.4$, Flashbot $\leq v1.11.5$, bloXroute, BSC $\leq v1.3.8$, core-geth$\leq v1.12.18$	✓	✓		✓	✗	✗			
Geth $< v1.11.4$, EigenPhi	✗	✗	✗	✗	✗	✗			
go-opera $\leq v1.1.3$		✗	✗	✗	✗	✗			
Erigon $\leq v2.42.0$				✗					
Besu $\geq v22.7.4$	✓	✗		✗					
Besu $< v22.7.4$	✗	✗		✗					
Nethermind $\geq v1.18.0$	✓			✗			✓		
Nethermind $< v1.18.0$	✗			✗			✗		
Reth $\geq v0.1.0-alpha.6$				✗				✓	
Reth $< v0.1.0-alpha.6$								✗	
OpenEthereum $\leq v3.3.5$				✗					✗

Other clients, including the three PBS clients and three clients on Ethereum-like networks, are mostly forks of the Geth clients except for minor code changes[4], and on them, MPFUZZ discovered 13 bugs of a similar nature to those found on Geth (of the historical and latest versions), as seen in Table 3.

We describe the patterns of these bugs as follows.

6.1 Found Eviction Attacks

Exploit XT_1: Direct eviction by future transactions: In this attack, given a mempool's initial state storing normal transactions, the attacker sends future transactions at high *GasPrice* to evict the normal transactions. This exploit is essentially the DETER-X attack [28].

In practice, Geth ($\leq v1.11.4$), Nethermind, Besu, and EigenPhi are vulnerable under this exploit, as in Table 3.

Exploit XT_2: Direct eviction by latent overdraft transactions: In this eviction attack, the attacker sends latent-overdraft transactions at high *GasPrice* to evict the normal transactions initially stored in the target mempool. These transactions are sent from k accounts, each of which sends l transactions, denoted by $k \times l$. The intention is to evade the limit on the number of transactions per sender. The evasion increases the attacker cost from one pending transaction to multiple. This exploit is essentially the DETER-Z attack [28].

In practice, Geth ($\leq v1.11.4$), Besu, EigenPhi, and go-opera are found vulnerable under this exploit.

Exploit XT_3: Compositional direct eviction (by combining XT_1 and XT_2): In some Ethereum clients, notably Geth, the limit on the number of transactions per sender is triggered under the condition that the mempool stores enough pending transactions (e.g., more than 5120 transactions in Geth). This eviction attack combines XT_1 and XT_2 to avoid triggering the condition and evade the protection. It works by maximizing the eviction of mempool under XT_1 until it is about to trigger the condition. It then conducts XT_2 by sending latent overdraft transactions under one sender, that is, $1 \times l$. Compared to XT_2, the combined exploit XT_3 achieves lower costs.

In practice, Geth ($\leq v1.11.4$) is vulnerable under this exploit where its mempool of capacity of 6144 slots triggers the limit of 16 transactions per sender when there are more than 5120 pending transactions. That is, XT_3 is configured with $l = 5120$. Other clients including EigenPhi and go-opera are also vulnerable.

Exploit XT_4: Indirect eviction by valid-turned-overdraft transactions: This eviction attack works in two steps: 1) The attacker first sends valid transactions at high *GasPrice* to evict normal transactions initially stored in the mempool. These transactions are sent from k accounts, each of which sends l transactions. 2) She then sends k transactions; each of the transactions is of nonce 1, at a high *GasPrice*, of high value v_1, and from the same k accounts in Step 1). The transactions would replace the transaction of the same sender and nonce sent in Step 1). Once the replacement is finished, they turn their child transactions into latent overdraft. Specifically, if a sender's balance is bal and the Ether value of transaction of nonce 2 is v_2, then v_1 is carefully crafted to enable turned latent overdraft, that is, $v_1 < bal$ and $v_1 + v_2 > bal$.

In practice, many clients are found vulnerable under this exploit, including Geth $< v1.11.4$, Erigon, Besu, Nethermind, OpenEthereum, EigenPhi, and go-opera.

Exploit XT_5: Indirect eviction by valid-turned-future transactions: This eviction attack works in two steps: 1) The attacker first sends valid transactions at high fees to evict normal transactions initially stored in the mempool. These transactions are sent by $k \times l$, that is, from k accounts, each with l transactions. For each sender, the transaction of nonce 1 has a fee, say f_1, slightly lower than the transactions of other nonces, that is, $f_1 < f_2$. 2) She then sends k transactions; each of the transactions is from a distinct sender from those used in Step 1) and of fee f' that $f_1 < f' < f_2$. The intent is that the k transactions in Step 2) evict the transactions of nonce 1 sent in Step 1), turning other child transactions sent in Step 1) into future transactions.

Exploit XT_6: Compositional indirect eviction (by multi-round valid-turned-future transactions): This exploit is an adaptive attack to the Geth $\geq v1.11.4$. Geth $\geq v1.11.4$ is patched against XT_{1-2} and adopts the following admission

[4] For instance, go-opera adopts its own fix against XT_1 based on the fork of Geth $< v1.11.4$.

policies: The mempool of $m' = 6144$ slots admits up to $py'_1 = 1024$ future transactions. When containing more than $py'_3 = 5120$ pending transactions, the mempool starts to limit that no more than $py'_2 = 16$ pending transactions from the same sender can be admitted.

Exploit XT_6 works in three steps: 1) it first evicts the initial mempool of normal transactions with $m'/py'_2 = 384$ sequences, each of 16 transactions sent from one unique sender, 2) it then sends $py'_1/(py'_2 - 1) = 65$ transactions to evict the parent transactions and creates $py'_1 = 1024$ future transactions in the mempool, 3) it sends one sequence of 5120 transactions from one sender to evict the normal transactions sent in Step 1), and 4) it sends one transaction to evict the parent transaction in Step 3, leaving the mempool of just one valid transaction. Step 3) can succeed because the precondition (w.r.t. $py'_3 = 5120$) of limiting transactions of the same sender does not hold.

In practice, Geth of all versions and the Geth forks, including Flashbot, bloXroute, BSC, and Ethereum Classic are found vulnerable under XT_6.

Exploit XT_7: Reversible evictions: Suppose a mempool of state st admits an arriving transaction tx by evicting an existing transaction tx', transitioning its state to st'. A mempool is reversible if one sends tx' to state st', and the mempool admits tx' by evicting tx, transitioning its state back to st.

The mempool on Nethermind $< v1.18.0$ can be reversible: Minimally, suppose a two-slot mempool stores tx_1 of low *GasPrice* from one sender and tx_2 of medium *GasPrice* from another sender $A2$ and receives tx_3 as a child of tx_1 and of high *GasPrice*. For instance, $tx_1 \left[\begin{smallmatrix} A & 1 \\ 1 & 1 \end{smallmatrix}\right], tx_2 \left[\begin{smallmatrix} B & 1 \\ 1 & 3 \end{smallmatrix}\right], tx_3 \left[\begin{smallmatrix} A & 1 \\ 2 & 5 \end{smallmatrix}\right]$. The Nethermind $< v1.18.0$ mempool would admit tx_3 and evict tx_2. After that, if one resends the evicted tx_2 back to the mempool storing tx_1 and tx_3, the Nethermind $< v1.18.0$ mempool would admit tx_2 and evict tx_3, looping back to its initial state.

An attacker observing reversible mempool mounts Exploit XT_7 to evict the mempool while bringing down attack costs. On Nethermind $< v1.18.0$, the attack works in two steps: 1) She first sends k transaction sequences from k senders,[5] each of l transactions (i.e., $k \times l$). In each l-transaction sequence, the nonce-1 transaction has a low *GasPrice*, say f_1, while all its child transactions are of high *GasPrice*, evicting normal transactions initially stored in the mempool. 2) The attacker sends $k \times (l-1)$ transactions from $k \times (l-1)$ new senders, all with *GasPrice* slightly higher than f_1. The mempool would admit these transactions to the mempool by evicting all child transactions sent in Step 1), bringing down the attack costs.

6.2 Found Locking Attacks

Exploit XT_8: Locking FIFQ: Certain mempool design prohibits eviction by admitting transactions as a FIFQ. Transactions are admitted only when there are empty slots. A full mempool always declines an arriving transaction. In practice, Reth $v0.1.0 - alpha.4$ adopts the FIFQ mempool design.

While a FIFQ mempool can not be vulnerable to eviction attacks, it can be easily locked by Exploit XT_8 as follows: Whenever the attacker observes an empty slot present in the mempool, she sends a pending transaction of a minimally necessary fee to occupy the slot. Any transactions that arrive after will encounter a full mempool and are declined.

Exploit XT_9: Locking mempool of no turning: Because of the risk of evicting a parent transaction that could lead to turning, certain mempool is designed to restrict eviction victims to child transactions; that is, only the transaction of maximal nonce under a given sender can be evicted. For instance, given an arriving transaction tx, the OpenEthereum mempool finds an existing child transaction tx' with a lower *GasPrice* than tx and evicts it to admit tx.

In Exploit XT_9, the attacker observes empty slots in the mempool (e.g., created by arriving blocks) and sends the same number of transactions to occupy them. The transactions are sent from one account where the child transaction has a higher *GasPrice* f_1 than normal transactions, while all its parent transactions have minimal fees. A normal transaction that arrives subsequently is declined because f_1 is higher than a normal transaction. Because each of its parent transactions has a low *GasPrice*, the mempool is locked at a low cost.

6.3 Evaluation of Found Turning Exploits

This and next subsections show the evaluation of turning attacks and locking attacks, and due to the space limit, the evaluation of other ADAMS attacks is deferred to Appendix D.

We design experiments to measure the success of ADAMS attacks in real Ethereum networks.

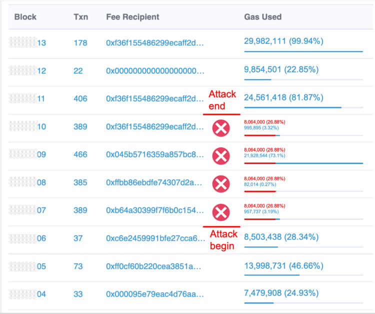
6.3.1 Evaluation in Testnet

Experimental settings: We set up our experiment platform by connecting the attacker node to the Goerli testnet. The attacker node runs an instrumented Geth client at the execution layer which sends crafted transactions to the testnet.

To monitor transaction propagation, we launch an independent supernode (called measurement node) in the same testnet. The measurement node runs a reconfigured Geth client where the limit of peers/neighbors is removed and can connect to as many neighbors as possible. The measurement node is not (directly) connected to the attacker node. When setting up our experiments, we ran the measurement node in Goerli for seven days and found the node is stabilized at 290 neighbors.

Experiment results: The instrumented measurement node is able to log the received messages from different neighbors; these messages include those of transactions and of transaction hashes. The measurement node could receive the same transaction from different neighbors, and the log stores the transaction-neighbor pairs. We started logging the received messages one month before the experiment.

[5] Nethermind does not limit transactions per sender, thus $k = 1$ in actual attacks.



Block	Txn	Fee Recipient	Gas Used
[redacted]13	178	0xf36f155486299ecaff2d...	29,982,111 (99.94%)
[redacted]12	22	0x000000000000000000...	9,854,501 (22.85%)
[redacted]11	406	0xf36f155486299ecaff2d...	24,561,418 (81.87%)
[redacted]10	389	0xf36f155486299ecaff2d...	8,064,000 (26.88%) 995,895 (3.32%)
[redacted]09	466	0x045b5716359a857bc8...	8,064,000 (26.88%) 21,928,544 (73.1%)
[redacted]08	385	0xffbb86ebdfe74307d2a...	8,064,000 (26.88%) 82,014 (0.27%)
[redacted]07	389	0xb64a30399f7f6b0c154...	8,064,000 (26.88%) 957,737 (3.19%)
[redacted]06	37	0xc6e2459991bfe27cca6...	8,503,438 (28.34%)
[redacted]05	73	0xff0cf60b220cea3851a...	13,998,731 (46.66%)
[redacted]04	33	0x000095e79eac4d76aa...	7,479,908 (24.93%)

Figure 3: Mounting XT_4 attacks on Goerli: Etherscan screenshot of the blocks generated during the attack

To do an experiment, we make the attacker node send XT_4 transactions using 384 accounts. Each account sends 16 valid pending transactions, followed by a replacement transaction. In total, the attacker node sends 6144 valid transactions and 384 replacement transactions. The *GasPrice* of the valid and replacement transactions are set to be 8 Gwei and 130 Gwei, respectively. Finally, we wrap up the experiment by waiting after the attacker node sends all messages and all replacement transactions are included in the blockchain.

Figure 3 shows the generated blocks in the experiment. We took the screenshot from `etherscan.io` and labeled it in red with information regarding the attack. Before the attack begins, the testnet normally utilizes $24-47\%$ of the Gas in a block. For ethics, our attacks are short (lasting four blocks).

Right after the launch of the attack, in block $xx07$, the Gas used by normal transactions drops to 3.19% and the Gas used by adversarial transactions (denoted by red bars in Figure 3) is 26.88%. The included adversarial transactions are 384 replacement transactions sent in the second round of XT_4. The $6144-384=5760$ child transactions sent in the first round are not included in the block. Other blocks during the attack, namely $xx08$ and $xx10$, are similar. In the third block $xx09$, 73.1% Gas is spent on including 82 normal transactions.To explain it, we inspect the raw history of transactions arriving at and logged by our measurement supernode and found that out of the 82 included normal transactions, only 4 are present in the log, implying the other 78 normal transactions are private ones that were not broadcast to the measurement node. Notice that our attacks require no discovery of critical nodes as needed in [28].

6.4 Evaluation of Found Locking Exploits

6.4.1 Exploit Extension

Mempool in Reth $v0.1.0-alpha.4$: Recall that the mempool is a FIFO queue: Any transaction residential in the mempool is never evicted, no matter how high an arriving transaction's fee is. Besides, when there is an empty slot in its mempool, it requires that the transaction admitted must have *GasPrice* higher than the latest block's *BasePrice*.

Actual exploit XT_{8a}: In XT_8, the attack cost increases with the block *BasePrice*. We manually propose a method to decrease the block *BasePrice* in Ethereum by mounting an eviction attack on the previous block. Specifically, in Ethereum (after EIP1559), given a recently produced block i, the block *BasePrice* of block $i+1$ is calculated dynamically as follows:

$$BasePrice(i+1) = \quad BasePrice(i)*[\frac{7}{8}+\frac{1}{4}*\frac{GasUsed(i)}{BlockLimit}] \tag{10}$$

Therefore, if an eviction attack can persistently lower the Gas utilization of recent blocks, the current *BasePrice* can be reduced, which decreases the costs of locking attacks on the current block. In Exploit XT_{8a}, we mount a series of eviction attacks first and then a locking attack against the Reth node.

Specifically, suppose in a network of a Reth node and a block validator node, the XT_{8a} attacker first keeps sending eviction attacks directly to the validator node for several consecutive blocks. When observing the block *BasePrice* drop sufficiently, the attacker mounts the regular locking attack XT_8 against the Reth node.

6.4.2 Attack Evaluation on Reth

Experimental settings: We set up an experiment platform for evaluating locking attacks, on which an attacker node is connected to a victim non-validator node, which is also connected to a workload generator node and a victim validator node. The attacker node also maintains a direct connection to the victim validator. The network topology is depicted in Figure 4. The non-validator node runs the Geth $v1.11.4$ client.

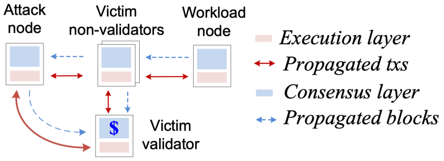


Figure 4: Experimental setup for locking attacks on Reth

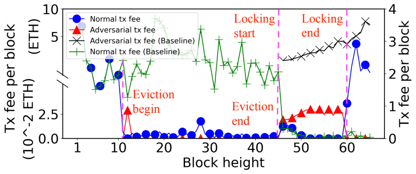


Figure 5: Evaluation of locking attack XT_8 on Reth

Evaluation of XT_8: To evaluate XT_8, we set up the workload generator that sends normal transactions to the validator Geth node. The non-validator Reth node has not yet joined the network. We let the attacker node first send eviction attacks (XT_6) directly to the victim validator for 35 consecutive blocks until it observes the block *BasePrice* drops sufficiently to 1 Gwei, which occurs at the 45-th in our experiment. The Reth node then joins the network with an empty mempool. The attacker node mounts a locking attack XT_8 with transactions of price 5 Gwei to occupy the Reth node's mempool.

We report the total fees of normal transactions included in the blocks – the lower the fees are, the more successful the locking attack is. We also report the adversarial transaction fees as the attack cost shown in Figure 5. When the locking attack begins at the 45-th block, the normal transaction fees increase because of the empty mempool on the Reth node which accepts some normal transactions in addition to the adversarial locking transactions. Then, after three blocks, the normal transaction fees quickly drop to near-zero Ether, showing the success of locking attacks. Interestingly, unlike the eviction attacks that cause low Gas utilization per block, the locking attacks just reduce the total Ether fees without reducing Gas utilization (i.e., the blocks produced under locking attacks from height 45 to 60 use the Gas of almost 100% block limit). Due to the high Gas utilization, block *BasePrice* keeps increasing during the locking attack. When reaching the 60-th block, the *BasePrice* is higher than the maximal transaction price tolerable by the attacker. In our experiment, the locking attacker stops at 60-th block, and the normal transaction fees immediately recover to the "normal" level. In practice, the attacker can repeat sending eviction attacks to reduce the *BasePrice* before mounting the locking attack again.

7 Evaluation of MPFUZZ

7.1 Performance

Preliminary of GoFuzz: Upon each fuzzing iteration, GoFuzz [16] generates a bit-string of variable length, feeds it to our fuzzer code, and receives from our fuzzer a binary value of feedback, be either positive or negative. The GoFuzz further checks if this run increases the code coverage. If the feedback is positive and code coverage increases, GoFuzz adds the current bit-string to the seeds before running into the next iteration.

Baseline B1: Stateless fuzzer. We implement a stateless fuzzer in the GoFuzz framework described above. 1) Given a bit string generated by GoFuzz, our fuzzer code parses it into a sequence of transactions; the length of the sequence depends on the length of the bit string. It skips a bitstring that is too short. 2) The fuzzer sets up and initializes a Geth `txpool` instance with m normal transactions, each sent from a distinct account, of nonce 1, of value 1 Ether (lower than the account balance), and of *GasPrice* 3 wei. 3) The fuzzer sends the transaction sequence to the initialized mempool for execution. 4) It checks the mempool state after execution: If the test oracle is met, it emits the current transaction sequence as an exploit.

Baseline B2: Concrete-state-coverage fuzzer. We build the second baseline, a mempool fuzzer that takes concrete state coverage as feedback. Specifically, in each iteration, the fuzzer appends a new transaction to the current transaction sequence. Given the m-slot mempool, the fuzzer tries m values for senders, nonces, *GasPrice*, and Ether amount. After sending the current transaction sequence to the mempool, it sorts the transactions in the mempool by senders and nonces. It then hashes the sorted transactions in the mempool and checks the presence of the hash digest in the seeds. If no hash exists, the current transaction sequence would increase the concrete-state coverage and is inserted to the corpus. The corpus is a FIFO queue where the seed inserted earlier has high priority to be de-queued.

Baseline B3: Invalid-tx energy fuzzer. Baseline B3 is similar to B2 except that it uses as the energy the number of invalid transactions on a mempool state. The seed with more invalid transactions in the mempool has higher energy or priority to be selected for the next round of fuzzing.

Baseline B4: feedback of no promising states: We design Baseline B4 for an ablation study on the effectiveness of using promising states as feedback in improving fuzzing performance. B4 is the same with MPFUZZ except that it removes the state promising-ness from its feedback. Specifically, the feedback formula in B4 includes only state coverage (*st_coverage*) and excludes state promisingness (*st_promising* as in Equation 7).

We implement Baseline B1 in Go/GoFuzz and implement B2, B3 and B4 in Python. We use a test oracle that can detect DETER attacks, that is, the number of invalid transactions in a mempool state equal $m-1$.

Experimental settings of fuzzing: We run the four baselines and MPFUZZ against a MUT in a Geth client reconfigured under two settings: A small setting where the mempool is resized to 6 slots $m = 6$ and all fuzzers run for two hours (if test oracle is not triggered), and a medium setting where $m = 16$ and fuzzers run for 16 hours. The fuzzing experiments are run on a local machine with an Intel i7-7700k CPU of 4 cores and 64 GB RAM.

Table 4: Fuzzing Geth $v1.11.3$'s mempool (in minutes) by different approaches to detect Exploit XT_3.

Settings	B1	B2	B3	B4	MPFUZZ
6slot-2h	Timeout	54	8	1.22	0.03
16slot-16h	Timeout	Timeout	447	Timeout	0.06

Results: Table 4 reports the results under the small and medium MUT settings. For the small MUT, B1 cannot find any DETER attack in two hours. B2, B3 and B4 can find XT_3 in 54 minutes, 8 minutes and 1.22 minutes respectively. By contrast, MPFUZZ finds the same exploit in 0.03 minutes.

For the medium setting, with 16 slots, baselines B1, B2, and B4 cannot find any DETER exploit in 16 hours, and B3 can find Exploit XT_3 in 7.4 hours. By contrast, MPFUZZ finds the same exploit in under a minute.

The evaluation on different clients is in Appendix C.

7.2 True/False Positive Rates

This section evaluates the true/false positives of the short exploits found by MPFUZZ. Recall that MPFUZZ runs against a small mempool (MUT) and may introduce false positives, that is, the exploits that satisfy bug oracles in MUT but do not in the actual larger mempool. We formally define true and false positives as follows:

Definition 7.1 (TP/FP exploits) *Consider a short exploit found by* MPFUZZ *on a MUT,* $\langle st_0, dc_0\rangle, ops \Rightarrow \langle st_n, dc_n\rangle$*, and a (manually) extended exploit on an actual mempool,* $\langle st'_0, dc'_0, ops' \Rightarrow \langle st'_n, dc'_n\rangle$*.*

The pair of exploits constitute a true positive eviction attack w.r.t. ε (λ)*, iff.* $st_0 \cap st_n = \varnothing \wedge asym_E(st_0, ops) < \varepsilon \wedge \; st'_0 \cap st'_n = \varnothing \wedge asym_E(st'_0, ops') < \varepsilon$ *.*

The pair of exploits constitute a false positive eviction attack w.r.t. ε (λ)*, iff.* $st_0 \cap st_n = \varnothing \wedge asym_E(st_0, ops) < \varepsilon \wedge \; (st'_0 \cap st'_n \neq \varnothing \vee asym_E(st'_0, ops') > \varepsilon)$*.*

True/false positives on locking attacks can be similarly defined from bug-oracle definitions 4.2 and 4.3.

We evaluate the false positives of the exploits found by MPFUZZ under varying ε and λ. Specifically, we design a two-step experiment: First, given a MUT, we run MPFUZZ with varying ε (and λ). Increasing ε allows MPFUZZ to find more short exploits; for each distinct short exploit, we manually extend it to an actual exploit. Second, we evaluate the extended exploits on an actual mempool (in the same local experimental setting as described in § D.1).

Table 5: True/false positives (TP/FP) of eviction and locking attacks discovered with varying ε and λ. ✓means satisfied.

Exploit	ε	$m = 16$ (MUT)		Default m		TP?
		Eq. 1	$asym_E$	Eq. 1	$asym'_E$	
Geth-XT_1	10^{-4}	✓	0	✓	0	✓
Geth-XT_3	.09	✓	0.083	✓	0.0002	✓
Geth-XT_6	.125	✓	0.125	✓	0.0003	✓
Geth-XT_2	0.2	✓	0.167	✓	0.0698	✓
Geth-XT_4	0.23	✓	0.208	✓	0.0768	✓
Geth-XT_5	0.23	✓	0.208	✓	0.0768	✓
Nethermind-XT_1	10^{-4}	✓	0	✓	0	✓
Nethermind-XT_4	0.11	✓	0.104	✓	0.0008	✓
Nethermind-XT_7	0.36	✓	0.355	✓	0.0012	✓
Besu-XT_2	0.2	✓	.167	✓	0.0754	✓
Besu-XT_4	0.23	✓	.208	✓	0.0083	✓
Erigon-XT_4	0.23	✓	0.208	✓	0.0894	✓

Exploit	λ	$m = 16$ (MUT)		Default m		TP?
		Eq. 4	$asym_D$	Eq. 4	$asym'_D$	
Reth-XT_8	0.34	✓	0.34	✓	0.015	✓
OpenEthereum-XT_9	0.46	✓	0.46	✓	0.0439	✓

Table 5 presents the details of exploits discovered by MPFUZZ. For instance, when setting *epsilon* at 0.0001 to find eviction attacks on Geth, MPFUZZ discovered one short exploit XT_1 with $asym_E = 0$ on MUT, which can be extended to an actual exploit on Geth mempool (of $m = 6144$) with $asym'_E = 0 < 0.0001 = \varepsilon$. This makes it a true positive (marked by ✓in the table). Increasing ε to 0.23 leads to discovering all six exploits XT_{1-6} on Geth. All these exploits are true positives. The table also includes the results of eviction exploits found on other clients and locking exploits. Overall, the true-positive rate of MPFUZZ remains at 100% for $\varepsilon \leq 0.36$ and for $\lambda \leq 0.46$ across Ethereum clients.

8 Discussions

Root causes of the exploits: We attribute all found exploits XT_{1-9} to four distinct causes: 1) the presence of certain admission patterns (including $XT_1/XT_2/XT_3/XT_4/XT_6/XT_7$). For instance, XT_4 allows transaction replacement to cause latent overdrafts in place of valid transactions. 2) the absence of certain admission patterns to cause mempool locking (including XT_9 and XT_8). For instance, the cause of XT_8 on Reth is that Reth mempool is a FIFO queue, and it disallows transaction eviction of any kind, which is risky. 3) Inconsistency across multiple admission patterns, notably XT_7. In XT_7, the cause is that a mempool allows an eviction and its reversed eviction at the same time. 4) Evadable conditions to context-sensitive admission patterns, including XT_3 and XT_6. Both exploits work on Geth, where the mempool triggers the limit of transactions per send under an evadable condition (when the mempool stores more than $py'_3 = 5120$ pending transactions).

Mitigation: We propose schemes to mitigate all newly discovered exploits comprehensively. In principle, given an exploit under a known cause, our mitigation design is to negate the cause. For instance, to mitigate XT_4, we propose declining transaction replacements that cause latent overdrafts. We have implemented this mitigation strategy on Geth, which was merged into the release of Geth $v1.11.4$ [8] . Similarly, to mitigate XT_7, we propose ensuring the consistency between eviction events and reversed events (e.g., if admitting tx_1 by evicting tx_2 is allowed, then admitting tx_2 by evicting tx_1 is not allowed). XT_1/XT_2 are mitigated by disallowing eviction of valid transactions by future/latent-overdraft transactions. XT_3/XT_6 are mitigated by removing the triggering condition of transaction sender limit (i.e., making $py'_3 = 0$). XT_8 can be mitigated by re-enabling transaction evictions in certain cases, such as price-based transaction eviction (i.e., a pending transaction arriving later and of higher price can evict another pending transaction of lower price and admitted earlier).

If an exploit relies on multiple causes (e.g., XT_3 on Causes 1 and 4), negating one cause suffices to mitigate the exploit.

There are more sophisticated cases where one exploit relies on the presence of an admission pattern, and another exploit relies on the absence of the pattern. For instance, XT_5 relies on the presence of an admission policy named SAP_5 that turns child transactions into future ones (as in Geth), and the success of XT_9 relies on the absence of the same policy, that is, $\neg SAP_5$ (as in OpenEthereum).

To mitigate both XT_5 and XT_9, we propose a defense strategy in which the admission decision regarding SAP_5 is non-deterministic or randomized so that SAP is not always on or off, making it hard for either XT_5 or XT_9 to always succeed.

Responsible bug disclosure: We have disclosed the discovered ADAMS vulnerabilities to the Ethereum Foundation, which oversees the bug bounty program across major Ethereum clients, including Geth, Besu, Nethermind, and Erigon. We also reported the found bugs to the developers of Reth, Flashbot, EigenPhi, bloXroute, BSC, Ethereum Classic, and Fantom. Besides 7 DETER bugs that MPFUZZ rediscovered, 24 newly discovered ADAMS bugs are reported. As of March 2024, 15 bugs are confirmed: XT_1 (Nethermind, EigenPhi), XT_2 (EigenPhi), XT_3 (EigenPhi), XT_4 (Geth, Erigon, Nethermind, Besu, EigenPhi), XT_5 (Geth), XT_6 (Geth, EigenPhi and Flashbot builder), XT_7 (Nethermind), and XT_8 (Reth). After our reporting, XT_4/XT_7/XT_8 have been fixed on Geth $v1.11.4$/Nethermind $v1.21.0$/Reth $v0.1.0-alpha.6$. Bug reporting is documented [20].

Ethical concerns: When evaluating attacks, we only mounted attacks on the testnet and did so with minimal impacts on the tested network. For instance, our attack lasts a short period of time, say no more than 4 blocks produced. We did not test our attack on the Ethereum mainnet. We also mask part of the block numbers in the screenshots (e.g., Figure 3) to prevent detailed attack inspection and reproduction. When reporting bugs, we disclose to the developers the mitigation design tradeoff and the risk of fixing one attack by enabling another attack (described above). To prevent introducing new bugs, we did not suggest fixes against turning-based locking (XT_5, XT_6) and locking (XT_8).

9 Conclusion and Future Works

Conclusion: This paper presents MPFUZZ, the first mempool fuzzer to find asymmetric DoS bugs by exploring symbolized mempool states and optimistically estimating the promisingness of an intermediate state in reaching bug oracles. Running MPFUZZ on popular Ethereum clients discovers new mempool-DoS vulnerabilities, which exhibit various sophisticated patterns, including stealthy mempool eviction and mempool locking.

Limitations and future works: The exploit generation in this work is not fully automated. Manual tasks include mempool reduction, MPFUZZ setup (configuring ϵ and symbols), exploit extension, etc. Automating these tasks is the future work.

MPFUZZ's bug oracles neither captures complete dependencies among concrete transactions nor guarantees completeness in finding vulnerabilities. Certified mempool security with completeness is also an open problem.

MPFUZZ targets a victim mempool of limited size: the vulnerabilities found in this work are related to transaction eviction, which does not occur in a mempool of infinite capacity. Thus, the MPFUZZ workflow cannot find exploits in the mempool of infinite capacity. It is an open problem whether a mempool of large or infinite capacity has DoS vulnerabilities.

10 Acknowledgments

The authors thank USENIX Security reviewers and Kangjie Lu for the discussion in the early stage of this work. Yibo Wang, Wanning Ding, Zhihua Yang, and Yuzhe Tang were partially supported by two Ethereum Foundation academic grants and NSF grants CNS-2139801, CNS-1815814, DGE-2104532. Kai Li is partially supported by NSF grant CNS-2347486 and one Ethereum academic grant.

References

[1] bloxroute builder. https://github.com/bloXroute-Labs/builder-ws.

[2] Bsc client. https://github.com/bnb-chain/bsc.

[3] core-geth. https://github.com/etclabscore/core-geth.

[4] Eigenphi builder. https://github.com/eigenphi/builder.

[5] Erigon. https://github.com/ledgerwatch/erigon.

[6] Flashbots mev-boost block builder. https://github.com/flashbots/builder.

[7] Geth: the go client for ethereum. https://www.ethereum.org/cli#geth.

[8] Geth v1.11.4. https://github.com/ethereum/go-ethereum/releases/tag/v1.11.4.

[9] go-opera. https://github.com/Fantom-foundation/go-opera/tree/master.

[10] Hyperledger besu. https://www.hyperledger.org/use/besu.

[11] Nethermind ethereum client. https://nethermind.io/client.

[12] Parity ethereum is now openethereum: Fast and feature-rich multi-network ethereum client. https://www.parity.io/ethereum/.

[13] Reth: Modular, contributor-friendly and blazing-fast implementation of the ethereum protocol. https://github.com/paradigmxyz/reth.

[14] Irreversible transactions: Finney attack. https://en.bitcoin.it/wiki/Irreversible_Transactions#Finney_attack, Retrieved July, 1, 2022.

[15] Memoria 700 million stuck in 115,000 unconfirmed bitcoin transactions. https://www.ccn.com/700-million-stuck-115000-unconfirmed-bitcoin-transactions/, Retrieved July, 1, 2022.

[16] Go fuzzing. https://go.dev/security/fuzz/, Retrieved May 31, 2023.

[17] Known attacks - ethereum smart contract best practices. https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-block-gas-limit, Retrieved May, 5, 2021.

[18] Report: Bitcoin (btc) mempool shows backlogged transactions, increased fees if so? https://goo.gl/LsU6Hq, Retrieved May, 5, 2021.

[19] Transaction pricing mechanism eip1559. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md, Retrieved Nov. 20, 2021.

[20] Flashbots block builder. https://sites.google.com/view/mpfuzz, Retrieved Oct, 2023.

[21] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In IEEE Symposium on SP 2017, pages 375–392, 2017.

[22] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In Kevin R. B. Butler and Kurt Thomas, editors, 31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022, pages 3255–3272. USENIX Association, 2022.

[23] Khaled Baqer, Danny Yuxing Huang, Damon McCoy, and Nicholas Weaver. Stressing out: Bitcoin "stress testing". In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers, volume 9604 of Lecture Notes in Computer Science, pages 3–18. Springer, 2016.

[24] Vitalik Buterin. Eip150: Gas cost changes for io-heavy operations.

[25] 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 2023 2023 IEEE Symposium on Security and Privacy (SP) (SP), pages 2517–2532, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[26] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In Jaeyeon Jung and Thorsten Holz, editors, USENIX Security 2015, Washington, D.C., USA, pages 129–144. USENIX Association, 2015.

[27] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. As strong as its weakest link: How to break blockchain dapps at RPC service. In 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021. The Internet Society, 2021.

[28] Kai Li, Yibo Wang, and Yuzhe Tang. DETER: denial of ethereum txpool services. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021, pages 1645–1667. ACM, 2021.

[29] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. LOKI: state-aware fuzzing framework for the implementation of blockchain consensus protocols. In 30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023. The Internet Society, 2023.

[30] Yuval Marcus, Ethan Heilman, and Sharon Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. IACR Cryptology ePrint Archive, 2018:236, 2018.

[31] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. Bdos: Blockchain denial of service, 2019.

[32] Daniel Pérez and Benjamin Livshits. Broken metre: Attacking resource metering in EVM. In 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society, 2020.

[33] Muhammad Saad, Laurent Njilla, Charles A. Kamhoua, Joongheon Kim, DaeHun Nyang, and Aziz Mohaisen. Mempool optimization for defending against ddos attacks in pow-based blockchain systems. In IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019, pages 285–292. IEEE, 2019.

[34] Muoi Tran, Inho Choi, Gi Jun Moon, Anh V. Vu, and Min Suk Kang. A Stealthier Partitioning Attack against Bitcoin Peer-to-Peer Network. In To appear in Proceedings of IEEE Symposium on Security and Privacy (IEEE S&P), 2020.

[35] Aviv Yaish, Kaihua Qin, Liyi Zhou, Aviv Zohar, and Arthur Gervais. Speculative denial-of-service attacks in ethereum. Cryptology ePrint Archive, Paper 2023/956, 2023. https://eprint.iacr.org/2023/956.

[36] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In Angela Demke Brown and Jay R. Lorch, editors, 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021, pages 349–365. USENIX Association, 2021.

A Observations Motivating MPFUZZ Design

(In)feasibility of code-coverage-only stateless fuzzing: To fuzz a mempool, a baseline design is to use a code-coverage-based fuzzer. In each iteration, the fuzzer generates a bit string, parses it into a sequence of transactions and sends them to the tested mempool for execution. Upon each end state, it checks whether the test oracle defined above is satisfied. This design takes code coverage as the only feedback and is stateless because the feedback does not consider the end state.

The stateless design would be ineffective in fuzzing mempool or finding ADAMS exploits. For example, consider fuzzing a three-slot mempool to find a DETER-X exploit consisting of three future transactions [28]. Suppose the current mempool input is a sequence of one future transaction. After mutation, it may try an input of two future transactions, which, however, does not increase code coverage (the second future transaction is executed by the same code path in a mempool as the first future transaction). Thus, the fuzzer discards the two-future-transaction input and misses finding the three-future-transaction exploit. We validate the ineffectiveness of stateless fuzzer by experiments in § 7.

A more promising design is stateful mempool fuzzing, in which the mempool state is included in the feedback to guide next-iteration fuzzing. In the previous example of finding a three-future-transaction exploit (DETER-X), sending two future transactions leads to a different mempool state than sending one. A simple state-coverage-based fuzzer would view this as positive feedback and further explore this direction toward finding the three-future-transaction exploit.

(In)feasibility of existing consensus fuzzers: Existing consensus fuzzers [25, 29, 36] cannot detect mempool DoS or ADAMS bugs. Specifically, Loki's test oracle is detecting system crashes, while a successful ADAMS attack does not necessarily trigger system crashes. Using Loki to detect ADAMS would cause many false negatives.

Fluffy detects the unsynchronizable difference of post-consensus states as vulnerability. However, the mempools in two clients, say Geth and Besu under benign transactions, would be different. In other words, post-consensus state differences across clients could indicate insecurity, but the difference in pre-consensus states across clients, like mempool, does not mean insecurity. Using Fluffly to detect ADAMS could cause false positives.

Tyr's test oracle detects property violations on post-consensus states. Particularly, while Tyr models state liveness, their definition "valid transactions should be executed eventually" is, unfortunately, over-simplifying and fails to model the legitimate pre-consensus cases in real Ethereum clients; for instance, valid transactions of low price can be dropped by Ethereum mempools and will not be eventually executed. Besides, they model double-spending transactions as the only type of invalid transaction. In Ethereum, double-spending transactions are of the same nonce. And there are many more sophisticated invalid transactions in Ethereum that Tyr does not model, including future transactions, latent overdrafts, etc.

We analyze the case of existing consensus fuzzers in detecting known DETER attacks [28]. Loki cannot detect DETER because DETER does not cause system crashes. Fluffy does not model future transactions and can not find the DETER attacks that rely on future transactions. Similarly, Tyr that does not model sophisticated invalid transactions would miss detecting DETER attacks.

A.1 Further Rationale of Symbolization

We now describe the design rationale of our symbolization technique.

- The first observation is that *real-world mempools admit transactions based on the symbolic value, not concrete value, of nonce and Ether amount*. Our design reflects this intuition. For instance, symbols of valid transactions, e.g., $\mathcal{P}, \mathcal{C}, \mathcal{N}$, are instantiated by MPFUZZ to a minimal amount 1 Ether. As long as the transferred value does not exceed the sender balance, transactions remain valid (i.e., no overdraft). Admission decisions remain the same for two valid transactions despite their difference in nonces or values.

  Likewise, Symbol $\mathcal{F}$ is instantiated by MPFUZZ to fixed nonce $n+1$: As long as the nonce is non-consecutive, no matter what specific value it takes (e.g., 3 or $m+1$), the mempool would deem it as a future transaction and makes the same admission decision.

  As a side note, the nonce of Symbol $\mathcal{F}$ is fixed at a large nonce $n+1$ (we denote it by far future transactions) instead of a small nonce, say 5 (we denote it by near future transactions), because near future transactions' nonce could be reconnected as consecutive and lead to duplicated states.

- The second observation is that *restricting eviction/replacement victims to only parent transactions can facilitate finding exploits quickly*. Specifically, while any transactions can be evicted/replaced, MPFUZZ evicts/replaces only parent transactions; such a strategy can cause maximal damage (i.e., the most child transactions turned) while saving the search space. Our symbol design reflects this idea. For instance, Symbol $\mathcal{R}$ covers any transaction of the same sender and nonce as a transaction in state st, but it is instantiated by MPFUZZ to only the transaction of nonce 1 (i.e., replacing a parent transaction).

B Case Study: How MPFUZZ Finds Exploits

We describe how MPFUZZ finds an exploit by presenting a case study on finding XT_6 in the latest Geth $\geq v1.11.4$.

Mempool reduction: Recall that a Geth mempool has a capacity of $m' = 6144$ slots, and its transaction-admission policies are characterized by three essential parameters: admitting up to $py'_1 = 1024$ future transactions and limiting up to $py'_2 = 16$ pending transactions from any senders when more than $py'_3 = 5120$ pending transactions are residing in the mempool. We set up the MUT to run the same codebase or the same admission policy but with smaller parameters: $m = 3, py_1 = 1, py_2 = 2, py_3 = 2$. What follows is a description of how MPFUZZ find a short exploit on this MUT.

Fuzzing: How MPFUZZ automatically finds exploits: Initially, the mempool is filled with $m = 3$ normal transactions. That is, the initial symbolized state is $\mathcal{N}\mathcal{N}\mathcal{N}$. The seed corpus sdb initially contains an empty string. MPFUZZ retrieves the empty string and appends to it with different symbols $\mathcal{O}_0$, $\mathcal{C}_0$, $\mathcal{P}_0$. Because of the initial state $\mathcal{N}\mathcal{N}\mathcal{N}$, only Symbol $\mathcal{P}_0$ is feasible. It generates the mutated input $\mathcal{P}$ and instantiates it to a parent transaction of a higher *GasPrice* than normal transactions (as described in § 5). Sending the input to the mempool gets the transaction admitted, leading to transitioned state $st_1 = \mathcal{N}\mathcal{N}\mathcal{P}$. This is a new state that is not in the corpus, and it evicts more normal transactions $\mathcal{N}$ than the previous state st_0; the input produces positive feedback, and the associated input-state pair $\langle P, st_1 = \mathcal{N}\mathcal{N}\mathcal{P}\rangle$ is added to the corpus.

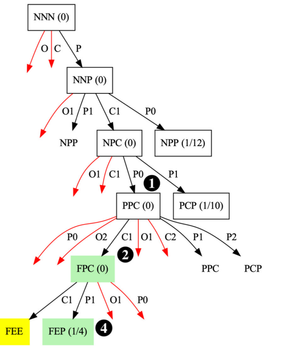


Figure 6: Snapshot of the MPFUZZ state-search tree when finding Exploit XT_6 on Geth v1.11.4.

Next, MPFUZZ retrieves from sdb a seed by high energy. According to Table 2, the energy of state $\mathcal{N}\mathcal{N}\mathcal{N}$ is $\frac{1}{3*3} * 0$, and the energy of state $\mathcal{N}\mathcal{N}\mathcal{P}$ is $\frac{1}{3*2+4} * 1 = 1/10 > 0$. Hence, seed $\mathcal{N}\mathcal{N}\mathcal{P}$ is selected. MPFUZZ tries input mutation and appends to the selected input $\mathcal{P}$ one of four new symbols, that is, $\mathcal{O}_1$, $\mathcal{C}_1$, $\mathcal{P}_0$ or $\mathcal{P}_1$. On state $\mathcal{N}\mathcal{N}\mathcal{P}$, 1) mutation transaction instantiated from symbol $\mathcal{O}_1$ is declined admission. 2) Mutation transaction from $\mathcal{C}_1$ is admitted, transitioning the state to $\mathcal{N}\mathcal{P}\mathcal{C}$, which produces positive feedback and is added to sdb. 3) Likewise, transaction $\mathcal{P}_0$ is admitted and produces state $\mathcal{N}\mathcal{P}\mathcal{P}$ of positive feedback; the mutated input is also added to sdb. 4) Mutation $\mathcal{P}_1$ is admitted but produces an identical state with mutation $\mathcal{P}_0$; thus, the mutated input is not added to sdb.

Now, there are four seeds in sdb: $\mathcal{N}\mathcal{N}\mathcal{N}(0)$, $\mathcal{N}\mathcal{N}\mathcal{P}(0)$, $\mathcal{N}\mathcal{P}\mathcal{C}(1/8)$, and $\mathcal{N}\mathcal{P}\mathcal{P}(1/12)$. In parentheses are their energy numbers. The seed of the highest energy $\mathcal{N}\mathcal{P}\mathcal{C}$ is selected (❶ in Figure 6). MPFUZZ then mutates $\mathcal{N}\mathcal{P}\mathcal{C}$ with 4 possibilities, that is, $\mathcal{O}_1$, $\mathcal{C}_1$, $\mathcal{P}_0$, and $\mathcal{P}_1$, which produce two end states with positive-feedback, that is, $\mathcal{P}\mathcal{P}\mathcal{C}$ and $\mathcal{P}\mathcal{C}\mathcal{P}$. They are added to the sdb with energy $\mathcal{P}\mathcal{P}\mathcal{C}(1/10)$ and $\mathcal{P}\mathcal{C}\mathcal{P}(1/10)$.

Let's say $\mathcal{P}\mathcal{P}\mathcal{C}(1/10)$ is selected (❷). MPFUZZ mutates input $\mathcal{P}\mathcal{C}_1\mathcal{P}_0$ with 7 mutation transactions, that is, $\mathcal{O}_1$, $\mathcal{O}_2$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\mathcal{P}_0$, $\mathcal{P}_1$, $\mathcal{P}_2$, which produces one end state with positive feedback, that is, mutation $\mathcal{C}_1$ is admitted and transitions state to $\mathcal{F}\mathcal{P}\mathcal{C}$. After that, $\mathcal{F}\mathcal{P}\mathcal{C}$ is the one with the highest energy and is chosen for the next-round fuzzing (❹). MPFUZZ mutates input $\mathcal{P}\mathcal{C}_1\mathcal{P}_0\mathcal{C}_1$ with 4 mutations and produces 2 state transitions with positive feedback. Upon state $\mathcal{F}\mathcal{P}\mathcal{C}$, mutation $\mathcal{C}_1$ is admitted and leads to state $\mathcal{F}\mathcal{E}\mathcal{E}$, which satisfies the bug oracle of eviction attacks under $\varepsilon = 0$. The algorithm then emits the found short exploit: $\langle st_0 = \mathcal{N}\mathcal{N}\mathcal{N}, dc_0 = \varnothing\rangle, ops = \mathcal{P}\mathcal{C}_1\mathcal{P}_0\mathcal{C}_1\mathcal{C}_1$ (recall Definition 4.1). The snapshot of the state-search tree is depicted in Figure 6.

Exploit extension: Given the short exploit automatically found on MUT (with $m = 3, py_1 = 1, py_2 = 2, py_3 = 2$), the next step is to extend it to a longer exploit functional on the actual Geth mempool (with $m' = 6144, py'_1 = 1024, py'_2 = 16, py'_3 = 5120$).

Exploit extension requires manual efforts: After identifying exploit $\langle st_0 = \mathcal{N}\mathcal{N}\mathcal{N}, dc_0 = \varnothing\rangle, ops = \mathcal{P}\mathcal{C}_1\mathcal{P}_0\mathcal{C}_1\mathcal{C}_1$ is unique, we extend it to the longer exploit by ensuring the same admission event occurs on the actual mempool as on the smaller MUT. In this process, it tries the next transaction of the same sender with the previous one but with an incremented nonce. If it fails, it switches to the next sender. It also tries transaction fees/prices based on measuring the fees of actual normal transactions, which may not be fixed as 3 in the MPFUZZ setting.

C Additional Eval. of MPFUZZ Performance

We further run our MPFUZZ on different clients, including Geth v1.10.11, Geth v1.11.4, Erigon v2.42.0, and Nethermind v1.18.0. The tested mempool is configured at 16 slots. We report the number of exploits found by MPFUZZ in a 16-hour period. Figure 7 presents the result that MPFUZZ finds 4096 short exploits in the first 8 hours on Geth-v1.10.11. On Geth-v1.11.4, MPFUZZ quickly finds one exploit (i.e., exploit XT_6 as described in § 6.1) within 2 minutes and does not find anymore in the next 16 hours. On Nethermind, it finds one exploit (exploit XT_1) within two minutes and finds the next one (exploit XT_4) near the end of 16-th hour. On Erigon, it finds one exploit (XT_4) in the 16-th hour.

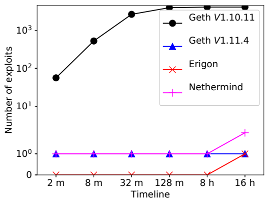


Figure 7: # exploits found

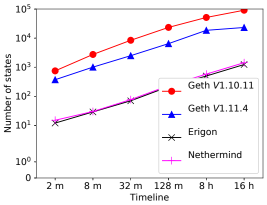


Figure 8: # states covered

Figure 8 shows the number of states that are explored by MPFUZZ on a 16-slot mempool in 16 hours. MPFUZZ explores 749 and 91428 states on Geth v1.10.11 in 2 minutes and 16 hours respectively. On Geth v1.11.4, MPFUZZ explores

372 and 23286 states in 2 minutes and 16 hours respectively. However, the performance of Erigon and Nethermind is much lower than that of Geth. MPFUZZ explores 12 and 1239 states on Erigon in 2 minutes and 16 hours respectively. On Nethermind, MPFUZZ explores 15 and 1433 states in 2 minutes and 16 hours, respectively. The reason MPFUZZ is more performant on Geth is that we implemented an external API on Geth to initialize the mempool; in each iteration of fuzzing on Geth, MPFUZZ calls the API to initialize the mempool. In contrast, on Nethermind and Erigon, MPFUZZ restarts the client in each iteration, which is consuming.

D Additional Attack Evaluation

D.1 Experiment on a Single Victim Node

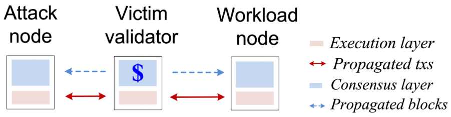


Figure 9: Experimental setup

Evaluation settings: Our goal is to evaluate the success rate and cost of different ADAMS attacks on a single victim node. Because some ADAMS attacks are sensitive to the workload of normal transactions, we first collect transaction workloads from the mainnet. Specifically, we instrumented a Geth client (denoted by Geth-m) to log every message it receives from every neighbor. The logged messages contain transactions, transaction hashes (announcements), and blocks. When the client receives the same message from multiple neighbors, it logs it as multiple message-neighbor pairs. We also log the arrival time of a transaction or a block.

Workload collection: We launched a Geth-m node in the mainnet on May 17, 2023, turned on logging for 5 hours, and collected the transactions propagated to it. We make the collected transactions replayable as follows: We use the account balances and nonces on the mainnet to set up the initial state locally. We then replace the original sender in the collected transactions with the public keys that we generated. By this means, we know the secret keys of transaction senders and are able to send the otherwise same transactions for experiments.

For experiments, we set up three nodes, an attack node sending crafted transactions, a workload node sending normal transactions collected, and a victim node receiving transactions from the other two nodes. The victim node is connected to both the attack and workload nodes. There is no direct connection between the attack node and the workload node. The attacker node runs an instrumented Geth $v1.11.4$ client that can propagate invalid transactions. The victim node runs the tested client. The workload node runs a vanilla Geth $v1.11.4$ client. On each node, we also run a Prysm $v3.3.0$ client at the consensus layer. The experiment platform is denoted in Figure 9. Among the three nodes, only the victim node is staked as a validator and would propose or produce blocks.

In each experiment, we first run the above "attacked" setup. We then run a "regular" setup that excludes the attack node. Under the regular setup, the workload node sends the normal transactions and blocks to the victim node. We compare the experiment results under the attacked setup and regular setup to show the success of the attack.

In the "attacked" setup, we collect the blocks produced and, given a block, we report two metrics: 1) total fees of benign transactions included, 2) total fees of attack transactions included. We also re-run the workload and victim nodes with the "regular" setup. We collect the blocks produced, and, given a block, we report two metrics: 3) total fees of transactions included, and 4) the cost of a baseline spamming attack with 100% success rate. For the latter, given a block, we select the transaction of the highest *GasPrice* and report the *GasPrice* multiplied by the Ethereum block Gas limit.

Table 6: Attack success rate and cost (Ether/block)

Clients	Exploit	Success rate	Cost	Baseline
Geth $v1.10.25$	XT_1	99.80%	0	11.39
	XT_2	99.42%	0.725	11.39
	XT_3	93.02%	0.0021	11.39
Geth $v1.11.4$	XT_4	99.42%	0.806	11.39
	XT_5	92.65%	0.806	11.39
	XT_6	94.74%	0.0022	11.39
Erigon $v2.42.0$	XT_4	92.53%	1.172	17.7
Nethermind $v1.18.0$	XT_4	99.60%	0.0021	10.75
	XT_7	84.63%	0.20	10.75
Besu $v22.7.4$	XT_2	99.63%	1.04	17.7
	XT_4	99.60%	1.06	17.7
Reth $v0.1.0\text{-}alpha.6$	XT_4	92.53%	0.672	17.6
Flashbot builder $v1.11.5$	XT_6	94.74%	0.0022	11.39
EigenPhi builder	XT_1	99.80%	0	11.39
	XT_2	99.42%	0.725	11.39
	XT_3	93.02%	0.0021	11.39
	XT_4	99.42%	0.806	11.39
	XT_6	94.74%	0.0022	11.39
bloXroute builder-ws	XT_6	94.74%	0.0022	11.39
go-opera $v1.1.3$	XT_2	99.12%	0.201	11.39
	XT_3	92.60%	0.0021	11.39
	XT_4	99.13%	0.221	11.39
	XT_6	93.76%	0.0022	11.39
BSC $v1.3.8$	XT_6	94.74%	0.0022	11.39
core-geth $v1.12.18$	XT_6	94.74%	0.0022	11.39
Reth $v0.1.0\text{-}alpha.4$	XT_8	99.04%	0.151	7.14
OpenEthereum $v3.3.5$	XT_4	99.56%	0.233	11.39
	XT_9	98.36%	0.177	18.35

The evaluation results of all attacks on all clients are in Table 6. On the six Ethereum clients in the public transaction path, the attack success rates are all higher than 84.63%, and the attack costs are all lower than 1.172 Ether per block, which is significantly lower than the baseline.

We also evaluate the attacks found on PBS and Ethereum-like clients. Under the same experimental settings as § D.1, the results show similar success rates and costs with the attacks on Geth (recall these clients are Geth forks). More specifically, Table 6 shows that the success rates are higher than 92.60%, and attack costs are lower than 0.806 Ether per block.