Asymmetric Mempool DoS Security: Formal Definitions and Provable Secure Designs

Wanning Ding
Department of Computer Science
Syracuse University
wding04@syr.edu

Yibo Wang
Department of Computer Science
Syracuse University
ywang349@syr.edu

Yuzhe Tang
Department of Computer Science
Syracuse University
ytang100@syr.edu

***Abstract*—The mempool plays a crucial role in blockchain systems as a buffer zone for pending transactions before they are executed and included in a block. However, existing works primarily focus on mitigating defenses against already identified real-world attacks. This paper introduces secure blockchain-mempool designs capable of defending against any form of asymmetric eviction DoS attacks. We establish formal security definitions for mempools under the eviction-based attack vector. Our proposed secure transaction admission algorithm, named SAFERAD-CP, ensures eviction-security by providing a provable lower bound on the cost of executing eviction DoS attacks. Through evaluation with real transaction trace replays, SAFERAD-CP demonstrates negligible latency and significantly high lower bounds against any eviction attack, highlighting its effectiveness and robustness in securing blockchain mempools.**

***Keywords*—Blockchain, mempool, asymmetric eviction DoS, defense**

I. Introduction

In public blockchains, a mempool is a data structure residing on every blockchain node, responsible for buffering unconfirmed transactions before they are included in blocks. On Ethereum, mempools are present in various execution-layer clients serving public transactions, such as Geth [1], Nethermind [2], Erigon [9], Besu [8], and Reth [10]. They are also utilized by block builders managing private transactions within the proposer-builder separation architecture, such as Flashbots [11], Eigenphi [12], and BloXroute builders [13].

Unlike conventional network stacks, mempools are permissionless and must accept transactions from unauthenticated accounts to maintain decentralization. This open nature, while essential for decentralization, makes the mempool vulnerable to denial-of-service (DoS) attacks. In such an attack, an adversary infiltrates the target blockchain network, establishes connections with victim nodes, and floods them with crafted transactions to deny mempool services to legitimate transactions. The disruption of mempool services can severely impact various blockchain subsystems, including block building, transaction propagation, blockchain value extraction (e.g., MEV searching), remote-procedure calls, and Gas stations. For instance, empirical studies have shown that disabling mempools can force the Ethereum network to produce empty blocks, undermining validators' incentives and increasing the risk of 51% attacks.

Related works & open problems: Denial of mempool services have been recently recognized and studied in both the research community and industry. In a large-scale blockchain, a mempool of limited capacity has to order transactions by certain priority criteria for transaction admission and eviction. The admission policy can be exploited to mount mempool DoS. The early attack designs [15], [24] work by sending spam transactions of high prices to evict benign transactions of normal prices. These attacks are extremely expensive and are not practical.

Of more realistic threat is the *Asymmetric* DeniAl of Mempool Service, coined by us as ADAMS attacks, where the attacker spends much less fees in the adversarial transactions he sends than the damage he caused, that is, the fees of evicted benign transactions that would be otherwise chargeable. DETER [20] is the first ADAMS attack studied in research works where the attacker sends invalid and thus unchargeable transactions (e.g., future transactions in Ethereum) to evict valid benign transactions from the mempool. The MemPurge attack [27] similarly evict Ethereum's (more specifically, Geths) pending-transaction mempool by crafting invalid overdraft transactions. These ADAMS attacks follow a single-step pattern and can be easily detected. In fact, DETER bugs have been fixed in Geth $v1.11.4$ [7], and there is a code fix implemented and tested against MemPurge on Geth [4].

More sophisticated and stealthy attacks are recently discovered by MPFUZZ, a symbolized stateful fuzzer for testing mempools [26]. The discovered attacks transition mempool states in multiple steps (see § VII-C for an example exploit, XT_6) and are stealthy to detection. They can evade the mitigation designed for earlier and simpler ADAMS attacks. For instance, Geth $v1.11.4$, which is already patched against DETER attacks, is found still vulnerable by MPFUZZ, such as under the XT_6 attack [26].

Current defenses against mempool DoS attacks typically involve patching vulnerabilities after specific attack patterns have been identified. This reactive approach, while necessary, cannot fully guarantee eviction security for the mempool, as there may be undiscovered attacks capable of bypassing these patches. Therefore, the key to ensuring robust mempool security lies in a formal understanding of mempool DoS security. Without a precise security definition, it is impossible to validate or certify the soundness or completeness of a mempool's defenses against unknown attacks, let alone design new mempools with provable security. Notably, the bug oracles used in tools like MPFUZZ optimize search efficiency but do not guarantee completeness, further highlighting the need for a comprehensive security framework.

Note that ADAMS attacks that exploit mempool admission

policies are not the only means to cause under-utilized blocks in a victim blockchain. There are other DoS vectors, notably the computing-resource-exhaustion attacks targeting any blockchain subsystems prior to block validation. Known exhaustion strategies include running under-priced smart-contract instructions [23], [16], [6] or exploiting "speculative contract execution capabilities, such as the `eth_call` in Ethereums RPC subsystem [19] or censorship enforcement in Ethereum PBS (proposer-builder separation) subsystem (i.e., the ConditionalExhaust attack in [27]). Resource exhaustion by adversarial smart contracts is out of the scope of this paper. Besides, denial of blockchain services have been studied across different layers in a blockchain system stack including eclipse attacks on the P2P networks [18], [21], [14], [25], DoS blockchain consensus [22], [3], DoS state storage [17], etc.

This work formulates the economic-security notions to protect mempools against asymmetric eviction DoS attacks. Economic security entails fee calculation, which may be non-deterministic in Ethereum and poses challenges in mempool designs. This work focuses on the mempool DoS by exploiting transaction admission policies, which normally rely only on deterministic transaction prices and are agnostic to non-deterministic Gas or fees.

Proposed policy: This work presents the first definitions of asymmetric eviction mempool DoS security and the provably secure mempool designs. Specifically, we conceive a general pattern of mempool eviction-based DoSes: Attackers aim to evict the victim transactions residential in a mempool. Based on the pattern, we formulate a security definition, that is, g-eviction security, which requires a secure mempool to lower bound the total fees of residential transactions under arbitrary sequences of transaction arrivals.

This work presents SAFERAD-CP, a transaction admission policy for securing mempools against asymmetric eviction DoSes. SAFERAD-CP can achieve eviction security by lower-bounding the fees of any arriving transactions causing eviction which is much higher than original Geth. We focus on designing admission policy in a pending-transaction mempool, that is, the pool is supposed to store only valid transactions (whereas future or other invalid transactions are buffered in a separate pool).[1]

SAFERAD-CP prevents valid residential transactions from being turned into invalid ones by evicting only "child-less" transactions, that is, the ones with the maximal nonce of its sender. Upon each arriving transaction, if the transaction is admitted by evicting another transaction, it enforces that the fee of the admitted transaction must not be lower than a pre-set lower bound.

Evaluation: We analyze SAFERAD-CP and prove the eviction-based DoS security with eviction bound: lower-bound the attack costs (i.e., the adversarial transaction fees) under eviction DoSes.

We implement a SAFERAD-CP prototype over Geth by evicting only "child-less" transactions. We collect real-world transactions from the Ethereum mainnet, and replay them against the SAFERAD-CP prototype. The evaluation shows SAFERAD-CP incurs a revenue change between $[+1.18\%, +7.96\%]$ under replayed real-world transaction histories. SAFERAD-CP's latency in serving transactions is at most 7.3% different from a vanilla Geth client. Under the attacks that latest Geth is known to be vulnerable for, SAFERAD-CP increases the attacker's cost by more than 10^4 times.

Contributions: Overall, this paper makes the following contributions:

- *New security definitions*: Presented the first economic-security definitions of mempools against asymmetric eviction DoSes. Specifically, we formulate a general pattern of mempool eviction DoSes and present a security definition to mitigate this attack pattern.

- *Proven secure designs*: Presented the first eviction DoS-secure mempool designs, SAFERAD-CP, that achieves proven eviction-based security. The security stems from SAFERAD-CP's design in lower-bounding the attack cost under eviction DoSes.

- *Performance & utility evaluation*: Implemented a SAFERAD-CP prototype on Geth and evaluated its performance and utility by replaying real-world transaction traces. It shows SAFERAD-CP incurs negligible overhead in latency and maintains a high lower bound on validator revenue under any adversarial traces.

II. Background and Notations

Transactions: In Ethereum, a transaction tx is characterized by a $sender$, a $nonce$, a $price$, an amount of computation it consumes, $GasUsed$, and the $data$ field relevant to smart-contract invocation. Among these attributes, transaction $sender$, $nonce$, and $price$ are "static" in the sense that they are independent of smart contract execution or the context of block validation (e.g., how transactions are ordered). This work aims at lightweight mempool designs leveraging only static transaction attributes. We denote an Ethereum transaction by its $sender$, $nonce$, and $price$. For instance, a transaction tx_1 sent from Account A, with nonce 3, of price 7 is denoted by $\langle A3, 7\rangle$.

A transaction tx_1 is tx_2's ancestor or parent if $tx_1.sender = tx_2.sender \wedge tx_1.nonce < tx_2.nonce$. We denote the set of ancestor transactions to transaction tx by $tx.ancestors()$. Given the transaction set in a mempool state ts, tx's ancestor transactions in ts and with consecutive nonces to tx are denoted by set $tx.ancestors() \cap ts$. For instance, suppose transactions tx_1, tx_2, tx_3, tx_4 are all sent from Alice and are with nonces 1, 2, 3, and 4, respectively. Then, $tx_4.ancestors() \cap \{tx_1, tx_3\} = \{tx_3\}$.

A transaction tx is a future transaction w.r.t. a transaction set ts, if there is at least one transaction $tx' \notin ts$ and tx' is an ancestor of tx. Given any future transaction tx in set ts, we define function $\textsc{isFuture}(tx, ts) = 1$.

Transaction fees: A transaction tx's fee is the product of $GasUsed$ and $price$, that is, $tx.fee = GasUsed \cdot price$. $GasUsed$ is determined by a fixed amount (21000 Gas) and the smart contract execution by tx. In Ethereum, The latter factor is sensitive to various runtime conditions, such as how transaction tx is ordered in the blocks. After EIP-1559, part of

[1] In the rest of the paper, the term mempool refers only to the pool storing pending transactions.

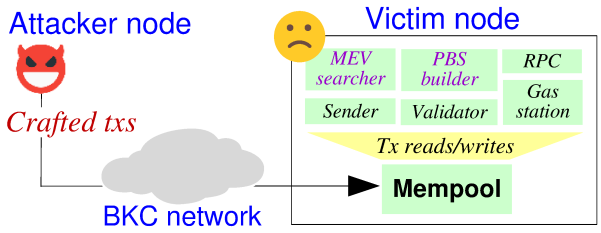


Fig. 1: Threat model of a victim mempool: In blue are downstream operators that rely on reading or writing the mempool. In dark blue are the operators in the private transaction path.

price and fees are burnt; in this case, block revenue excludes the burnt part.

The notations used in this paper (some of which are introduced later in the paper) are listed in Table I. Notably, we differentiate the unordered set and ordered list: Given an unordered transaction set, say ops, an ordered list of the same set of transactions is denoted by a vector, $\vec{ops}$. A list is converted to a set ops by the function $unorder(\vec{ops})$, and a set is converted to a list $\vec{ops}$ by the function $toOrder(ops)$.

TABLE I: Notations: txs means transactions.

	Meaning		Meaning
ops	Arriving txs	$\vec{ops}$	List of arriving txs
st	Txs in mempool	$\vec{st}$	List of txs in mempool
dc	Txs declined or evicted from mempool	m	Mempool length
$\langle A3, 7\rangle$	A tx of sender A, nonce 3, and price 7		

Blockchain mempools: In blockchains, recently submitted transactions by users, a.k.a., unconfirmed transactions, are propagated, either privately or publicly, to reach one or multiple validator nodes. Mempool is a data structure a blockchain full node uses to buffer the "unconfirmed" transactions before these transactions are included in blocks.

In Ethereum 2.0, transaction propagation follows two alternative paths. A public transaction is propagated to the entire network, while a private transaction is forwarded by a builder to the proposers who he has established the connection with. In both scenarios, the mempool faces the same design issues: Because the mempool needs to openly accept a potentially unlimited number of transactions sent from arbitrary EOA accounts, it needs to limit the capacity and enforce policies for transaction admission. In practice, we found the mempool storing public transactions has the same codebase as that storing private mempool.[2]

III. Threat Model

In the threat model, an adversary node is connected, either directly or indirectly, to a victim node on which a mempool serves various downstream operators reading or inserting transactions. Figure 1 depicts some example mempool-dependent operators, including a transaction sender who wants to insert her transaction to the mempool, a full node that reads the mempool to decide whether a received transaction should be propagated, a validator or PBS builder that reads the mempool and selects transactions to be included in the next block, an MEV searcher that reads the mempool to find profitable opportunities, a Gas-station service that reads the mempool to estimate appropriate Gas price for sending transactions, etc.

The adversary's goal is two-fold: 1) Deny the victim mempool's service to these critical downstream operators. This entails keeping all normal transactions out of the mempool so that the transaction read/write requests from operators would fail. 2) Keep the adversary's cost asymmetrically low. This entails keeping the transactions sent by the adversary away from being included in the blockchain.

The adversary has the capacity to craft transactions and send them to reach the victim mempool. In the most basic model, the adversary is directly connected to the victim node. In practice, the adversary may launch a super-node connected to all nodes and aim at attacking all of them or selecting critical nodes to attack (as done in DETER [20]). Alternatively, the adversary may choose to launch a "normal" node connected to a few neighbors and propagate the crafted transactions via the node to reach all other nodes in the network.

IV. Security Definition

To start with, we characterize two mempool states at any time: the set of transactions residing in the mempool denoted by st, and the set of transactions declined or evicted from the mempool denoted by dc. We also characterize the list of confirmed transactions included in produced blocks by $\vec{bks} = \{\vec{b}\}$ and the list of transactions arriving by $\vec{ops}$.

A mempool commonly supports two procedures: transaction admission and block building. We specify the first one, which is relevant to this work.

Definition 4.1 (Tx admission): Given an arriving transaction tx_i at a mempool of state st_i, an admission algorithm ADTX would transition the mempool into an end state st_{i+1} by admitting or declining tx_i or evicting transactions te_i. Formally,

$$\text{ADTX}(st_i, tx_i) \rightarrow st_{i+1}, te_i \tag{1}$$

Definition 4.2 (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 list of arriving transactions in $\vec{ops}$, and ends up with an end state $\langle st_n, dc_n\rangle$. Then, a validator continually builds blocks from transactions in the mempool st_n until it is empty, leading to eventual state $\langle st_l = \varnothing, dc_l = dc_n\rangle$ and newly produced blocks $\vec{bks}_l$ with transactions in st_n. This transaction-admission timeline is denoted by $f(\langle st_0, \varnothing\rangle, \vec{ops}) \Rightarrow \langle st_n, dc_n\rangle$.

As in the above definition, this work considers the timeline in which block arrival or production does not interleave with the arrival of adversarial transactions. We leave it to the future work for interleaved block arrival and attacks. In the following, we define the mempool security against asymmetric attacks.

Definition 4.3 (Mempool eviction security): Consider any mempool that initially stores normal transactions st_0, receives a list of benign and adversarial transactions $\vec{ops}$ and converts to the end state st_n.

[2]For instance, the mempool in Flashbot builder [11] used to store both private and public transactions is a fork (with no code change) of the mempool storing only public transactions in Geth.

The mempool is secure against asymmetric eviction attacks, or g**-eviction-secure**, if.f. the total transaction fees under any adversarial transactions are higher than $g(st_0)$ where $g(\cdot)$ is a price function that takes as input a mempool state and returns a price value. Formally,

$$\begin{aligned} &\forall st_n, \vec{ops}, \exists \text{function } g(\cdot) \\ &\text{s.t., } \forall st_0, fees(st_n) \geq g(st_0) \end{aligned} \tag{2}$$

The g-eviction-security definition ensures that under arbitrary attacks, the total fees of transactions inside the mempool are lower bound by a value dependent only on normal transactions in initial state. This definition reflects the following intuition: If the normal transactions in initial state can provide enough block revenue for validators, the g-security of mempool ensures the mempool under any adversarial workloads (i.e., attacks) have enough fees or enough block revenue for validators.

Definition 4.4 (Mempool locking security): Consider under attacks in which the mempool of initial state storing only benign transactions $\langle st_0, \varnothing \rangle$ receives a transaction sequence $\vec{ops}$ interleaved of adversarial and benign transactions, transitions its state, and reaches the end state $\langle st_n, dc_n \rangle$, as shown Equation 3.

The mempool is said to be secure against asymmetric locking attacks, or h**-locking-secure**, if.f. the maximal price of transactions declined or evicted in the end state under attacks, i.e., dc_n, is lower than a certain price function $h()$ on the end-state mempool under attacks, i.e., $h(st_n)$. It is required that for any mempool state st, $h(st)$ must be lower than the average transaction price in st, namely $\forall st, h(st) < avgprice(st)$. Formally,

$$\begin{aligned} &\forall st_0, \vec{ops}, f(\langle st_0, \varnothing \rangle, \vec{ops}) \Rightarrow \langle st_n, dc_n \rangle \\ &\exists \text{function } h(\cdot) < avgprice(\cdot) \\ &\text{s.t., } maxprice(dc_n) < h(st_n) \end{aligned} \tag{3}$$

The h-locking-security definition ensures that the maximal price of declined or evicted transactions from the mempool is upper bound by that of the transaction staying in the mempool.

V. SAFERAD FRAMEWORK

A. Tx-Admission Algorithm of CP

The proposed Algorithm 1 enforces the invariant that each admitted transaction to a mempool evicts at most one transaction from the mempool.

The algorithm initially maintains a mempool of state st and receives an arriving transaction ta. The algorithm decides whether and how to admit ta and produces as the output the end state of the mempool and the evicted transaction te from the mempool. In case that ta is declined by the mempool, $te = ta$.

Internally, the algorithm first pre-checks the validity of ta on mempool st (in Line 1). If ta is a future transaction or overdrafts its sender balance, the transaction is deemed invalid and is declined from entering the mempool. The algorithm only proceeds when ta passes the validity precheck.

Algorithm 1 SAFERAD-CP(MempoolState st, Tx ta)

```
1:  if PRECKV(ta, st) == 0 then                ▷ Precheck tx validity
2:      return te = ta;                        ▷ Decline invalid ta
3:  end if
4:  tes⃗=TORDER(st.findChildless(), price);
5:  te=tes⃗.lastTx();
6:  if ||st|| == m then
7:      if ta.price ≤ te.price then
8:          return te = ta;                    ▷ Decline ta
9:      else
10:         st.admit(ta).evict(te);
11:     end if
12: else
13:     st.admit(ta);
14:     te =NULL;
15: end if
```

It then sorts all childless transactions in mempool ($st.findChildless()$) in descendant order based on tx's price. As described next, function $\vec{tes} = \text{TORDER}(st)$ produces a order of transactions in the mempool, denoted by $\vec{tes}$. It selects the last transaction on this ordered list, denoted by te. That is, te has the lowest score on $\vec{tes}$.

If the mempool is full and $ta.price \leq te.price$ (Line 7), the algorithm declines the arriving transaction ta, that is, $te = ta$ (Line 8). Otherwise, if the mempool is full and $ta.price > te.price$, the algorithm admits ta and evicts te (Line 10).

If the mempool is not full, the algorithm always admits ta to take the empty slot (Line 13).

B. Eviction Secure of CP

Policy CP achieves eviction security in the sense that it ensures the total prices monotonically increase.

Suppose a mempool runs Algorithm 1 and transitions from state st_i to st_{i+1}. No transaction in st_i can be turned into a future transaction in st_{i+1}. Because the eviction candidates can only be the childless transactions (Recall Line 4 in Algorithm 1).

Lemma 5.1 (Monotonic price-increasing): If a mempool runs Algorithm 1, the sum of transaction prices in the mempool monotonically increases, or the mempool is considered to be monotonic price-increasing. Formally,

$$\sum_{tx \in st_n} tx.price \geq \sum_{tx \in st_0} tx.price \tag{4}$$

Proof: Consider that a mempool running Algorithm 1 receives an arriving transaction ta_i and transitions from state st_i to st_{i+1}. We aim to prove that,

$$\sum_{tx \in st_{i+1}} tx.price \geq \sum_{tx \in st_i} tx.price \tag{5}$$

Now we consider three cases for state transition: 1) ta_i is declined, 2) ta_i is admitted by taking an empty slot in st_i (i.e., no transaction is evicted), and 3) ta_i is admitted by evicting te_i.

In Case 1), $st_i = st_{i+1}$. Thus,

$$\sum_{tx \in st_i} tx.price = \sum_{tx \in st_{i+1}} tx.price$$

In Case 2), $st_{i+1} = \{ta_i\} \cup st_i$. Thus,

$$\begin{aligned} \sum_{tx \in st_{i+1}} tx.price &= \sum_{tx \in st_i \cup \{ta_i\}} tx.price \\ &= \sum_{tx \in st_i} tx.price + ta_i.price \\ &\geq \sum_{tx \in st_i} tx.price \end{aligned}$$

In Case 3), we denote by st the set of transactions in both st_i and st_{i+1}. That is, $st = st_i \cap st_{i+1}$. In Case 3), we have $st_{i+1} \backslash \{ta_i\} = st_i \backslash \{te_i\} = st$. Applying Line 7 in Algorithm 1, we can derive the following:

$$\begin{aligned} \sum_{tx \in st_{i+1}} tx.price &= \sum_{tx \in st \cup \{ta_i\}} tx.price \\ &= \sum_{tx \in st} tx.price + ta_i.price \\ &\geq \sum_{tx \in st} tx.price + te_i.price \\ &= \sum_{tx \in st \cup \{te_i\}} tx.price \\ &= \sum_{tx \in st_i} tx.price \end{aligned}$$

Therefore, in all three cases, Equation 5 holds. In general, for any initial state st_0 and any end state st_n that is transitioned from st_0 with $i \in [0, n-1]$, one can iteratively apply Equation 5 for $i \in [0, n-1]$ and prove the sum of transaction price monotonically increases. ■

Theorem 5.2 (g-eviction security of Algorithm 1): A mempool running Algorithms 1 is g-eviction secure. That is, given a sequence of arriving normal and adversarial transactions $o\vec{p}s$ to the mempool state, the total transaction fees in the end-state mempool under attacks st_n are lower-bounded by the following:

$$\forall st_0, o\vec{p}s, fees(st_n) \geq g(st_0) \tag{6}$$

Proof: Due to Lemma 5.1, we prove the property of increasing sum of transaction prices during the mempool state transition, that is, Equation 4.

And the minimal gas for each transaction is 21000. We have a lower bound for transaction fee which is $21000 \cdot tx.price$, that is,

$$\begin{aligned} fees(\vec{st_n}) &\geq 21000 \cdot \textstyle\sum_{tx \in st_n} tx.price \\ &\geq 21000 \cdot \textstyle\sum_{tx \in st_n} tx.price \\ &\geq 21000 \cdot \textstyle\sum_{tx \in st_0} tx.price \end{aligned} \tag{7}$$

Equation 6 holds. ■

C. Locking Insecure of CP

Policy SAFERAD-CP is not locking secure. We illustrate this with a counterexample that highlights how an attacker can exploit policy CP to achieve a denial-of-service (DoS) with minimal cost.

Consider a mempool implementation that imposes no limitations on the number of transactions sent by a single sender. Suppose the mempool is at full capacity, and all transactions originate from a single sender. These transactions are represented as: $tx_1, tx_2, \ldots, tx_{n-1}$, each with a price of 1, while the only transaction without child dependencies, tx_n, has the highest nonce and a significantly higher price of 10,000.

The mempool operates under policy CP where an incoming transaction tx_a can only evict an existing transaction if tx_a has a higher price than the transaction it aims to evict. Given this policy, tx_a would need a price greater than 10,000 to evict tx_n. This creates a scenario where the attacker can effectively lock the mempool with a low-cost strategy by ensuring tx_n remains in place, thus preventing the eviction of other transactions priced at 1.

VI. Implementation Notes on Geth

We build a prototype implementation of SAFERAD-CP on Geth $v1.11.4$. We describe how the pending mempool in vanilla Geth handles transaction admission and then how we integrate SAFERAD-CP into Geth.

Background: Geth mempool implementation: In Geth $v1.11.4$, the mempool adopts the price-only Policy patched with extra checks. Concretely, upon an arriving transaction ta, ⓪ Geth first checks the validity of ta (e.g., ta is an overdraft), then it checks if the mempool is full. ① If so, it finds the transaction with the lowest price as the candidate of eviction victim te'. ② It then removes te' from the primary storage and the secondary index. ③ At last, it adds ta to the primary storage and the secondary index. If the mempool is not full, ④ Geth adds ta to the primary storage and to the secondary index.

For fast transaction lookup, Geth $v1.11.4$ maintains two indices to store mempool transactions (i.e., each transaction is stored twice): a primary index where transactions are ordered by price and a secondary index where transactions are ordered first by senders and then by nonces.

In Step ⓪, we adopt the patch against MemPurge attacks [4].

Implementation of SAFERAD-CP on Geth: Geth's mempool architecture is well aligned with Algorithm 1. We overwrite Step ① in Geth; instead of finding the transaction with the lowest price, we find the childless transaction with the lowest price, which is to scan Geth's price-based index from the bottom, that is, the transaction with the lowest price; for each transaction, we check if the transaction has a defendant in the mempool by querying the second index. If so, we continue to the transaction above in the primary index and repeat the check. If not, we select the transaction to be eviction victim te. Steps ②, ③, and ④ remain the same except that reference te' is replaced with te.

VII. Revenue Evaluation

A. Experimental Setups

Workload collection: For transaction collection, we first 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.

We ran a Geth-m node in the mainnet and collected transactions propagated to it from Sep. 5, 2023 to Oct. 5, 2023. In total, $1.5 * 20^8$ raw transactions are collected, consuming 30 GB-storage. We make the collected transactions replayable as follows: We initialize the local state, that is, account balances and nonces by crawling relevant data from infura.io. 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.

We choose 8 traces of consecutive transactions from the raw dataset collected, each lasting 2.5 hour. We run experiments on each 2.5-hour trace. The reason to do so, instead of running experiments directly on the one-month transaction trace, is that the initialization of blockchain state in each trace requires issuing RPC queries (e.g., against infura) on relevant accounts, which is consuming; for a 2.5-hour trace, the average time of RPC querying is about one day. To make the selected 2.5-hour traces representative, we cover both weekdays and weekends, and on a single day, daytime and evening times.

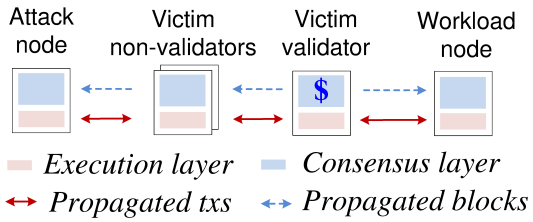


Fig. 2: Experimental setup

Experimental setup: For experiments, we set up four nodes, an optional attack node sending crafted transactions, a workload node sending normal transactions collected, a victim non-validator node propagating the transactions and blocks between attack node and victim validator node, and a victim validator node receiving transactions from the workload node and attack node through non-validator node. The victim validator node is connected to both the workload node and victim non-validator node which is connected to the attack node. There is no direct connection between the attack node and the workload node. The attack node runs an instrumented Geth $v1.11.4$ client (denoted by Geth-a) that can propagate invalid transactions to its neighbors. The victim nodes runs the target Ethereum client to be tested; we tested two victim clients: vanilla Geth $v1.11.4$ and SAFERAD-CP CP; the latter one is implemented as addon to Geth $v1.11.4$. 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 depicted in Figure 2. Among the four nodes, we stake Ether to the consensus-layer client on the victim validator node, so that only the victim validator node would propose or produce blocks.

We run experiments in two settings: under attacks and without attacks. For the former, we aim at evaluating the security of SAFERAD-CP under attacks, that is, how successful DoS attacks are on SAFERAD-CP. In this setting, we run all three nodes (the victim, attacker and workload nodes). For the latter, we aim at evaluating the utility of SAFERAD-CP under normal transaction workloads. In this setting, we only run victim and workload nodes, without running the attack node.

In each experiment, 1) we replay the collected transactions as follows: For each original transaction tx collected, we send a replayed transaction tx' by replacing its sender with a self-generated blockchain address. $GasUsed$ is simulated: If tx runs a smart contract, tx' does not run the same contract. Instead, we make tx' run our smart contract with $tx'.GasUsed$ equals $tx.GasUsed$ if tx is included in the mainnet blockchain, or equal $tx.Gas$ (i.e., the Gas limit set by the transaction sender) if tx is not included. 2) When replaying a collected block, we turn on the block-validation function in Prysm, let it produce and validate one block, and send the block (which should be different from the content of the collected block) to the Geth client. We then immediately turn off block validation before replaying the next transaction in the trace.

B. Revenue Under Normal Transactions

This experiment compares different mempool policies by evaluating their block revenue under the same transaction workloads.

Experimental method: We consider two mempool policies, the baseline one in Geth $v1.11.4$ and SAFERAD-CP on the baseline. Given each policy, we replay the 8 transaction traces in the same way as before and collect the produced blocks. We report the average revenue per block collected from the blocks.

Results: Figure 3 presents the revenue of the selected 150 consecutive blocks from the 600 blocks in Trace 2. The numbers of the three mempool policies are close, and they fluctuate in a similar way. Table II presents the aggregated results by the total revenue and revenue per block. Compared to Geth $v1.11.4$, Policy CP's revenue per block falls in the range of $[100\% + 1.18\%, 100\% + 7.96\%]$. This result suggests SAFERAD-CP incurs no significant change of block revenue under normal transactions.

TABLE II: Average block revenue (Ether) of different mempool policies. In bold are max and min numbers.

Trace	CP (Ether)	Geth (Ether)
1	0.73 (+3.13%)	0.71
2	1.17 (+6.97%)	1.09
3	0.85 (+4.55%)	0.81
4	0.54 (+4.56%)	0.52
5	0.78 (+6.66%)	0.73
6	1.19 (+4.23%)	1.14
7	**0.56 (+1.18%)**	0.55
8	**0.64 (+7.96%)**	0.59

C. Revenue Under Attacks

Background of attacks: This experiment evaluates the security of SAFERAD-CP against known attacks. Given that our

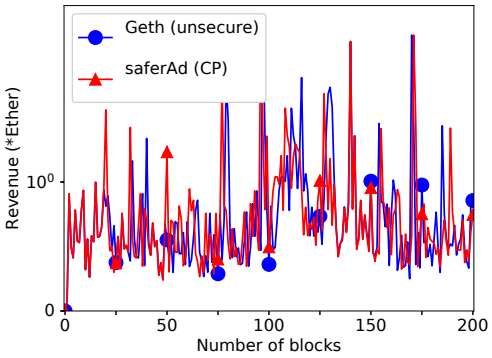


Fig. 3: Block revenue w/w.o. SAFERAD-CP under Trace 1.

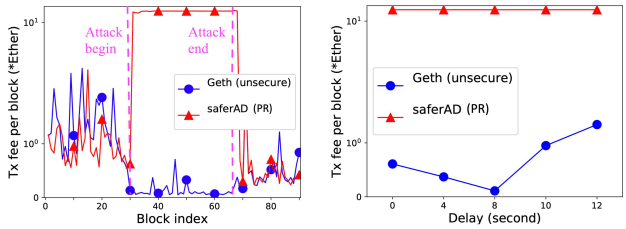
implementation is on Geth $v1.11.4$, we choose the attacks still effective on this version: XT_6 [26]. Briefly, the attack works in four steps: 1) It first evicts the Geth mempool by sending 384 transaction sequences, each of 16 transactions from a distinct sender. The transaction fees are high enough to evict normal transactions initially in the mempool. 2) It then sends 69 transactions to evict 69 parent transactions sent in step 1) and turn their child transactions into future transactions. 3) Since now there are more than 5120 pending transactions in the mempool, Geth's limit of 16 transactions per send is off. It then conducts another eviction; this time, it sends all 5120 transactions from one sender, evicting the ones sent in the previous round. 4) At last, it sends a single transaction to turn all transactions in the mempool but one into future transactions. The overall attack cost is low, costing the fee of one transaction.

Because XT_6 can evict Geth's mempool to be left with one transaction, we use as Geth's bound the maximal price in a given mempool times the maximal Gas per transaction (i.e., block Gas limit, namely 30 million Gas).

Experimental method: We set up the experiment platform described in § VII-C. In each experiment, we drive benign transactions from the workload node. Note that the collected workload contains the timings of both benign transactions and produced blocks. On the 30-th block, we start the attack. The attack node observes the arrival of a produced block and waits for d seconds before sending a round of crafted transactions.

The attack phase lasts for 36 blocks; after the 66-th block, we stop the attack node from sending crafted transactions. We keep running workload and victim nodes for another 24 blocks and stop the entire process at the 90-th block. We collect the blocks produced and, given a block, we report the total fee of transactions included.

Results: Figure 4a reports the metrics for XT_6 on two victim clients: vanilla Geth $v1.11.4$ and SAFERAD-CP. Before the attack is launched (i.e., before the 30-th block), the two clients produce a similar amount of fees for benign transactions, that is, around $0.7-2.0$ Ether per block. As soon as the attack starts from the 30-th block, Geth's transaction fees quickly drop to zero Ether, which shows the success of XT_6 on unpatched Geth $v1.11.4$. There are some sporadic spikes (under 0.7 Ether per block), which is due to that XT_6 cannot lock the mempool on Geth. Patching Geth with SAFERAD-CP can fix the vulnerability. After the attack starts on the 30-th block, the



(a) XT_6 w. 8-sec. delay (single node)(b) XT_6 w. varying delay (single node)

Fig. 4: Validator revenue under attacks: With and without SAFERAD-CP defenses.

transaction fees, instead of decreasing, actually increase to a large value; the high fees are from adversarial transactions and are charged to the attacker's accounts. The high fees show the effectiveness of SAFERAD-CP against known eviction-based attacks (XT_6).

Figure 4b shows the total fee per block under XT_6 with varying delays, where the delay measures the time between when a block is produced and when the next attack arrives. The results of Geth $v1.11.4$ show that with a short delay, the total transaction fees in mempool are high because XT_6 cannot lock a mempool, and the short block-to-attack delay leaves enough time to refill the mempool. With a median delay (e.g., sending an attack 8 seconds after a block is produced), the attack is most successful, leaving mempool at zero Ether. With a long delay, the in-mempool transaction fees grow high due to the attack itself being interrupted by the block production. Under varying delays, the transaction fees of the mempool practicing SAFERAD-CP policy would remain constant at a high value, showing the defense effectiveness against attacks of varying delays.

D. Estimation of Eviction Bounds

This experiment estimates the eviction bounds of different admission policies under real-world transaction workloads.

Experimental method: In the experiment, we replayed the transaction traces we collected in § VII-A each of two Ethereum clients, be it either CP, or vanilla Geth $v1.11.4$. In each run, right after producing each block, say bk_i, we record the mempool snapshot st_i. Then, assuming an attack starts right after the block bk_i is produced and lasts for the next 10 blocks, we estimate the lower bound of fees in the mempool right after block bk_{i+10} under arbitrary eviction attacks.

1) For CP, we use Equation 7 as the estimation bound. 2) For the baseline, we consider vanilla Geth $v1.11.4$ under XT_6; instead of mounting actual attacks (which is time-consuming), we estimate the attack damage by considering that the mempool under attack contains only one transaction, which is consistent with the mempool end-state under an actual XT_6 attack.

TABLE III: Average, 95-th, and 5-th percentile of eviction bounds under different policies on eight transaction traces

	Avg. bound (Ether)	95% bound (Ether)	5% bound (Ether)
CP	6.05	13.72	2.63
Geth	$0.63 \cdot 10^{-3}$	$1.05 \cdot 10^{-3}$	$0.17 \cdot 10^{-3}$

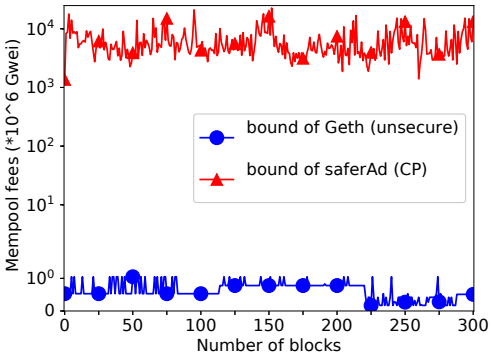


Fig. 5: Block revenue lower bounds.

Results: Figure 5 presents the results of estimated bounds over time. Compared to the mempool fees post attacks in Geth $v1.11.4$, both SAFERAD policies achieve high eviction bounds. The bound of CP is higher than that of Geth $v1.11.4$ by 4 orders of magnitude. Table III presents statistics of the estimated bounds on the two clients tested. It includes the average, 95-th, and 5-th percentile of eviction bounds. SAFERAD-CP achieves statistically higher bounds than the baseline of Geth. On average, CP's eviction bound is 6.05 Ether, which is much higher than the $0.63 \cdot 10^{-3}$ Ether in Geth under attacks. And specifically, for CP, 5% of its bounds exceed 13.72 Ether, and 95% exceed 2.63 Ether.

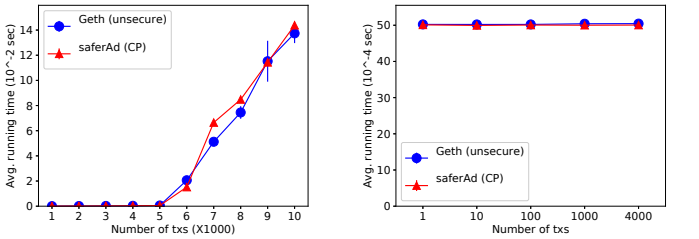
VIII. Performance Evaluation

A. Platform Setup

In this work, we use Ethereum Foundation's test framework [5] to evaluate the performance of the implemented countermeasures on Ethereum clients. Briefly, the framework runs two phases: In the initialization phase, it sets up a tested Ethereum client and populates its mempool with certain transactions. In the workload phase, it runs multiple "rounds", each of which drives a number of transactions generated under a target workload to the client and collects a number of performance metrics (e.g., latency, memory utilization, etc.). In the end, it reports the average performance by the number of rounds. In terms of workloads, we use one provided workload (i.e., "Batch insert") and implement our own workload ("Mitigated attacks").

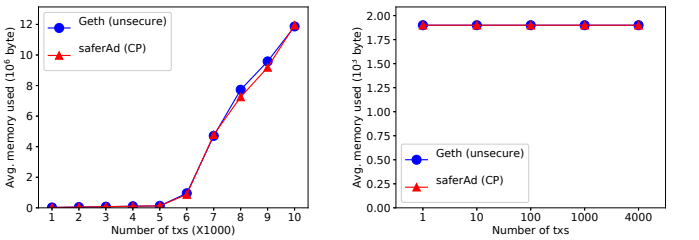
B. Experimental Results

Workload "Batch insert": We use the provided workload "Batch insert" that issues `addRemote` calls to the tested `txpool` of the Geth client. We select this workload because all our countermeasures are implemented inside the `addRemote` function. In the initialization phase, this workload sends no transaction to the empty mempool. In the workload phase, it runs one round and sends n_0 transactions from one sender account, with nonces ranging from 1 to n_0, and of fixed Gas price 10000 wei. When n_0 is larger than the mempool size, Geth admits the extra transactions to the mempool, buffers them, and eventually deletes them by running an asynchronous process that reorg the mempool (i.e., Function `pool.scheduleReorgLoop()` in Geth).



(a) Workload "Batch insert" (b) Workload "Mitigated attack TN1"

Fig. 6: Running time of SAFERAD-CP on Geth $v1.11.4$



(a) Workload "Batch insert" (b) Workload "Mitigated attack TN1"

Fig. 7: Memory usage of SAFERAD-CP on Geth $v1.11.4$

We report running time and memory usage in Figure 6a and Figure 7a, respectively. In each figure, we vary the number of transactions n_0 from 1000 to 10000, where $n_0 \in [1000, 5000]$ indicates a non-full mempool, and $n_0 \in [5000, 10000]$ indicates a full mempool. It is clear that on both figures, the performance overhead for a non-full mempool is much lower than that for a full mempool; and when the mempool is full, both the running time and memory usage linearly increase with n_0. The high overhead is due to the expensive mempool-reorg process triggered by a full mempool. Specifically, 1) upon a full mempool ($n_0 > 5000$), on average, Geth $v1.11.4$ hardened by SAFERAD-CP causes $1.073\times$ running time and $1.000\times$ memory usage, compared to vanilla Geth $V1.11.4$. 2) When the mempool is not full ($n_0 \leq 5000$), on average, Geth $v1.11.4$ hardened by SAFERAD-CP causes $1.017\times$ running time and $1.000\times$ memory usage, compared to vanilla Geth $V1.11.4$. Overall, the performance overhead introduced by SAFERAD-CP is negligible.

Note that Geth $v1.11.4$ that mitigates DETER causes $1.083\times$ running time compared to vanilla Geth $v1.11.3$, which is vulnerable to DETER.

Workload "Mitigated attack TN1": We additionally extend the framework with our custom workloads for countermeasure evaluation. Here, we describe the performance under one custom workload, "Mitigated attack TN1".

In the initialization phase, it first sends n_1 pending transactions to an empty mempool sent from n_1' accounts, each of which has transactions of nonces ranging from 1 to $\frac{n_1}{n_1'}$. All the parent transactions of nonce 1 are of Gas price $1,000$ wei, and the others are of $200,000$ wei. It then sends 1024 future transactions and $5120 - n_1$ pending transactions from different accounts, all with nonce 1, to fill the mempool. In the workload phase, it sends n_1' transactions, each of price $20,000$ wei, to evict the parents and turn children into future transactions.

In each experiment, we fix n_1' at 10 accounts and vary n_1 to be $10, 100, 1000$ and 4000. We run the workload ten times and report the average running time and memory used with the standard deviation, as in Figure 6b and Figure 7b. The running time and memory used are insensitive to the number of transactions (n_1). The performance overhead introduced by SAFERAD-CP is negligible.

REFERENCES

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

[2] Nethermind ethereum client. https://www.nethermind.io/nethermind-client.

[3] Irreversible transactions: Finney attack, Retrieved July, 1, 2022.

[4] Fixing mempurge attacks in geth v1.12.2 (6 lines of code from line 887 to line 894 is added). https://github.com/fs3l/go-ethereum-TNX-defense/tree/eb9bfb43705c2ab94088bb21b5bbf0c257720034, Retrieved Sep, 2023.

[5] txpool_test.go in geth. https://github.com/ethereum/go-ethereum/blob/master/core/txpool/txpool_test.go, Retrieved Mar. 3, 2023.

[6] 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.

[7] Geth v1.11.4 release note. https://github.com/ethereum/go-ethereum/releases/tag/v1.11.4, Retrieved July, 2023.

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

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

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

[11] Flashbot builder. https://github.com/flashbots/builder, Retrieved April, 2023.

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

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

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

[15] 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.

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

[17] Zheyuan He, Zihao Li, Ao Qiao, Xiapu Luo, Xiaosong Zhang, Ting Chen, Shuwei Song, Dijun Liu, and Weina Niu. Nurgle: Exacerbating resource consumption in blockchain state storage via mpt manipulation. arXiv preprint arXiv:2406.10687, 2024.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] Yibo Wang, Wanning Ding, Kai Li, and Yuzhe Tang. Understanding ethereum mempool security under asymmetric dos by symbolic fuzzing, 2023.

[27] 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.