Asymmetric Mempool DoS Security: Formal Definitions and Provable Secure Designs

Wanning Ding
Syracuse University
wding04@syr.edu

Yuzhe Tang ✉
Syracuse University
ytang100@syr.edu

Yibo Wang
Syracuse University
ywang349@syr.edu

***Abstract*—A mempool is a security-critical subsystem in a public blockchain. Recent mempool attacks, notably asymmetric DoS, have shown their ability to severely damage the Ethereum network. This paper tackles the open research problem of designing principled and non-intrusive defenses against asymmetric mempool DoSes with provable security. It presents the first mempool economic-security definitions based on mempool-observable conditions. It then presents SAFERAD, a framework of secure mempool designs with provable security against asymmetric DoSes. To defend against dual attacks by evicting and locking a victim mempool, SAFERAD adopts a non-trivial design of enforcing an upper bound of the attack damage under the locking attacks and a lower bound of the attack cost under the eviction attacks. With a prototype implementation on Geth and evaluation under real transaction traces, the results show SAFERAD has low overhead in latency and block revenue, implying non-intrusiveness and practicality.**

1. Introduction

In public blockchains, a mempool is a data structure residing on every blockchain node, and its job is to buffer unconfirmed transactions before they are included in blocks. On Ethereum, mempools are present in various execution-layer clients serving public transactions [5], [7], [3], [6], [8] and block builders serving private transactions, as in the PBS or proposer-builder separation architecture [4], [2], [1].

Unlike the conventional network stack, mempools are permissionless, and by design, they have to openly accept transactions sent from unauthenticated accounts. This open nature, while necessary for achieving decentralization, makes the mempool susceptible to denial-of-service (DoS) attacks. In such an attack, an adversary joins a target blockchain network, befriends victim nodes, and sends them crafted transactions with the goal of denying the mempool service to normal transactions. The failed service of a mempool can cripple a range of downstream blockchain subsystems such as block building, transaction propagation, blockchain value extraction (i.e., MEV searching), remote-procedure calls, Gas stations, etc. For instance, it has been empirically shown that by denying mempools, one can force the entire Ethereum network to produce empty blocks [25], stripping away validators' incentives and putting the blockchain at risk from 51% attacks.

Existing mempool DoS attacks are highly practical, achieving high success rates yet requiring low monetary costs from attackers. They are asymmetric in the sense that an attacker can inflict much higher damage to victims than the cost he needs to invest. DETER [25] is the first asymmetric mempool DoS that evicts normal transactions by crafting invalid transactions at high prices. MemPurge [32] similarly evicts pending transactions using a certain type of overdraft transactions. These attacks follow simple transaction patterns and can be easily detected; in fact, recent Ethereum clients like Geth $v1.11.4$ have been patched against DETER [11], and there is an open-source code patch against MemPurge [13]. Unfortunately, more mempool-DoS attacks have been recently discovered by a mempool fuzzer named MPFUZZ [31] on a wide range of Ethereum clients including the patched ones; these "second-generation" mempool DoS attacks are much more sophisticated and function by sending a multi-step sequence of crafted transactions to trigger a series of mempool-state transitions to inflict damage. Their transaction patterns are stealthy, and the full mitigation is an open research problem.

Security definitions: To end this arms race between attackers and defenders, the key lies in a formal understanding of the mempool DoS security. Without a precise security definition, it would be impossible to validate or certify the soundness or completeness of a mempool's defense against unknown attacks, let alone design new mempools with provable security. Note that the bug oracles in MPFUZZ [31] optimize search efficiency and don't guarantee completeness.

One may attempt to define mempool-DoS security directly on notions like adversarial transactions and transaction fees. However, these notions are unobservable to a mempool, as they are dependent on execution-time conditions; see § 4 for detail. We propose mempool-observable economic-security definitions for evaluating a mempool's defense against DoS. The idea is as follows: Given a mempool of any initial state and receiving an arbitrary sequence of arriving transactions, the security of the mempool relies on a lower bound of transaction fees inside the mempool and an upper bound of fees of transactions left outside the

✉Yuzhe Tang is the corresponding author.

mempool, either evicted or declined upfront. As discussed in § 4.1, the upper and lower bounds jointly ensure the failure of any asymmetric mempool DoS attacks, including the variants of eviction attacks and locking attacks [31]. Our security definitions are based only on mempool-observable conditions: They consider arbitrary transactions and are agnostic to the benign or adversarial nature of these transactions. Our fee bounds rely only on "static" transaction attributes that are observable to a mempool, such as senders, nonces, and prices, but not Gas.

Secure designs: Designing a secure mempool with both eviction- and locking-security poses challenges because mempool eviction and locking form dual attacks. On the one hand, if a mempool's admission policy is too loose, an adversary can exploit it to overly evict normal transactions. On the other hand, if the policy is too strict, the adversary can exploit it to lock the mempool to decline normal transactions arriving.

This work presents SAFERAD, a lightweight transaction admission framework that secures a mempool against *both* eviction- and locking-based DoSes. SAFERAD prevents the risky eviction that turns existing valid transactions into invalid ones; this is achieved by evicting only "childless" transactions without descendants. SAFERAD prevents locking the mempool at a low-cost state; this is done by enforcing an upper bound on the price of any declined transaction. These rules may force the mempool to sometimes exhibit counter-intuitive admission behavior, such as admitting a low-priced transaction to evict a high-priced transaction, leading to a drop in total fees. SAFERAD is designed to limit the effect of such spurious admission by ensuring the monotonic increase of a lower bound of fees. SAFERAD admits transactions based only on "static" transaction attributes without speculatively executing smart contracts and is lightweight, free of resource-exhaustion risks.

Evaluation: We prove the security of SAFERAD against eviction- and locking-attacks: SAFERAD can upper-bound the attack damage under locking attacks (i.e., the fees of victim transactions declined) and lower-bound the attack costs under eviction attacks (i.e., the fees of adversarial transactions included in blocks).

We implement a SAFERAD prototype over a variant of Geth $v1.11.4$ patched against DETER [25] and MemPurge [32], and our prototype maintains additional transaction indexes. We collect transaction traces from the Ethereum mainnet and replay them to evaluate the utility and performance of the SAFERAD prototype. The results show that compared to the vanilla, unsecured Geth $v1.11.4$, SAFERAD causes a reasonable change of block revenue between $[-1.33\%, +7.96\%]$ and incurs extra latency by at most 7.3%. Under the attacks that Geth $v1.11.4$ is known to be vulnerable for, SAFERAD increases the attacker's cost by more than 10000 times.

Contributions: This paper makes the following contributions:

• *New security definitions*: We presented the first mempool economic-security definitions to evaluate a mempool's defense against asymmetric DoS. The definitions are based on observable conditions in practical mempools and comprehensively cover both eviction- and locking-based attacks [31].

• *Provable-secure designs*: We proposed SAFERAD, a framework of secure mempool designs with provable security against asymmetric DoSes. We have proven SAFERAD's security against arbitrary asymmetric-DoS attacks — The security stems from SAFERAD's design in upper-bounding the attack damage under locking attacks and lower-bounding the attack cost under eviction attacks.

• *Performance & revenue evaluation*: We implemented a SAFERAD prototype on Geth and evaluated its performance and revenue. Under real-world transaction traces, SAFERAD shows a small overhead in latency and block revenue. Besides, SAFERAD shows a high lower bound under eviction attacks and a relatively low upper bound under locking attacks.

2. Background and Related Works

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 $\text{ISFUTURE}(tx, ts) = 1$.

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 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 1. Notably, we differentiate the unordered set and ordered list: Given an unordered transaction set, say as, an ordered list of the

same set of transactions is denoted by a vector, $\vec{as}$. A list is converted to a set as by the function $unorder(\vec{as})$, and a set is converted to a list $\vec{as}$ by the function TORDER(as).

TABLE 1: Notations: txs means transactions.

	Meaning		Meaning
as	Arriving txs	$\vec{as}$	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 buffers the "unconfirmed" transactions before these transactions are included in blocks. In Ethereum 2.0, transaction propagation follows two 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 cases, 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 transactions.*

2.1. Related Works

Mempool DoS: The early designs of mempool DoSes [18], [29] work by sending spam transactions of high prices to evict benign transactions of normal prices. These attacks are extremely expensive and are not practical.

A more realistic threat is the asymmetric denial of mempool service, where the attacker spends much less fees in the adversarial transactions he sends than the damage he caused, that is, the fees of evicted or declined benign transactions that would be otherwise chargeable. DETER [25] is the first asymmetric DoS 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 DETER vulnerability is used to construct an active measurement approach, TopoShot [24], to reveal the Ethereum network topology for the first time. The MemPurge attack [32] similarly evicts Ethereum's (more specifically, Geth's) pending-transaction mempool by crafting invalid overdraft transactions. The DETER bug is fixed in Geth $v1.11.4$ [10], and there is a tested code patch to mitigate MemPurge on Geth [13].

More sophisticated and stealthy attacks are recently discovered by MPFUZZ, a stateful symbolized fuzzer for testing mempools [31]. The discovered attacks are staged and stateful attacks in that they transition mempool states via multiple steps (see § 8.3 for an example exploit, XT_6).

*For instance, the mempool in Flashbot builder [4] used to store both private and public transactions is a fork (with no code change) of the mempool storing only public transactions in Geth.

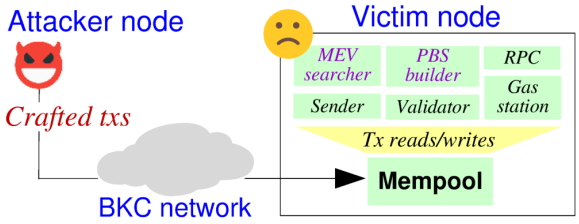


Figure 1: Threat model of a victim mempool: In blue are downstream operators that rely on reading or writing the mempool. In dark blue are the operators in the private transaction path.

Thus, they are stealthier and have successfully evaded the mitigation rules intended for DETER and MemPurge. 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 [31].

Resource exhaustion and blockchain DoS: Besides misusing transaction admission policies, there are other means to cause under-utilized blocks in a victim blockchain. One can aim to exhaust computing resources. Known strategies include running under-priced smart-contract instructions [28], [19], [14] or exploiting "speculative" contract execution capabilities, such as the `eth_call` in Ethereum's RPC subsystem [23] or censorship enforcement in Ethereum PBS (proposer-builder separation) subsystem (i.e., the ConditionalExhaust attack in [32]). Resource exhaustion by adversarial smart contracts is out of the scope of this paper. Besides, denial of blockchain services has been studied across different system layers including eclipse attacks on the P2P networks [22], [26], [17], [30], DoS blockchain consensus [27], [9], DoS state storage [21], etc.

3. Threat Model

Adversary's goals and capacities: 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 include 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 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 [25]). 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.

4. Economic-Security Definitions

Rationale: This work aims to develop principled defenses for blockchain mempools against asymmetric DoS (Denial of Service) attacks, including unknown ones. The key observation motivating this work is that, regardless of how adversarial transactions are crafted in an asymmetric DoS attack, the success of such an attack hinges on cost asymmetry. Specifically, the damage caused by the attack, measured by the fees of normal transactions excluded from produced blocks (due to denied mempool service), must be higher than the attack cost, measured by the fees of adversarial transactions included in produced blocks.

One can define mempool security directly by negating the above cost-asymmetry condition. That is, a secure mempool needs to ensure that, under all circumstances, the fees of normal transactions excluded from the mempool are lower than the fees of adversarial transactions included in the mempool (or blocks). However, such a formulation is impractical and based on conditions that a mempool cannot observe. For instance, in a permissionless setting, it is not possible to discern whether a transaction originates from an adversarial account or a benign one. Additionally, the fees a transaction charges, or more specifically, the computation or Gas an Ethereum transaction incurs, depend on the runtime context at the time of block validation (e.g., blockchain states and transaction ordering), which are not observable by a mempool. In practice, in most Ethereum clients, the smart-contract execution occurs after the mempool (e.g., to avoid resource-exhaustion risks [32]), leaving the dynamic transaction attributes like the Gas amount unobservable by the mempool.

To work around these design constraints, this work aims at establishing static bounds of transaction fees, instead of precisely estimating fees, based only on observable transaction attributes (e.g., prices, senders, and nonces) without relying on the unobservable Gas. It aims to bound a mempool's transaction fees under an arbitrary sequence of arriving transactions and initial states.

We propose economic security notions for mempools based on the following idea: Given a mempool of any initial state and receiving an arbitrary sequence of arriving transactions, the security of the mempool relies on 1) a lower bound of transaction fees inside the mempool, and 2) an upper bound of fees of transactions left outside the mempool, either evicted or declined upfront. As will be analyzed, the former quantifies the economic security of mempool under an abstract attack vector, eviction attacks, and the latter quantifies the economic security of mempool against locking attacks. Next, we specify the mempool processes, or timelines, before presenting the security definitions.

Specification: 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 $\vec{as}$.

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 ta_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 ta_i or evicting transactions te_i. Formally,

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

Definition 4.2 (Tx admission timeline). In a transaction admission timeline, a mempool is initialized at state $\langle st_0, dc_0 = \varnothing \rangle$, receives a list of arriving transactions in $\vec{as}$, 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 ordered transactions in st_n. This transaction-admission timeline is denoted by $f(\langle st_0, \varnothing \rangle, \vec{as}) \Rightarrow \langle st_n, dc_n \rangle$.

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_{0b} and receives a list of normal transactions $\vec{as}_b$, that is, the timeline without attack in Equation 3. Any adversary sends a list of adversarial transactions $\vec{as}_a$, at such a timing that adversarial transactions and normal transactions are interleaved to arrive at the following order: $\vec{as}_a \oplus \vec{as}_b$. That is the timeline under attack as in Equation 2.

The mempool is secure against asymmetric eviction attacks, or g**-eviction-secure**, if.f. the total transaction fees under arbitrary transaction ordering in the end state reached in the timeline under attack (Equation 2) are higher than $g(st_{nb})$ where st_{nb} is the end state of mempool in the timeline without an attack (Equation 3) and $g(\cdot)$ is a price function that takes as input a mempool state and returns a price value. Formally,

$$\forall st_{0b}, \vec{as_b}, \vec{as_a}, \vec{as} = as_a \oplus as_b,$$
$$f(\langle st_{0b}, \varnothing \rangle, \vec{as}) \Rightarrow \langle st_n, dc_n \rangle \quad (2)$$
$$f(\langle st_{0b}, \varnothing \rangle, \vec{as_b}) \Rightarrow \langle st_{nb}, dc_{nb} \rangle \quad (3)$$
$$\exists \text{function } g(\cdot) \text{ s.t., } \forall \vec{st_n}, fees(\vec{st_n}) \geq g(st_{nb}) \quad (4)$$

The g-eviction-security definition ensures that under arbitrary attacks, the total fees of transactions inside the mempool are lower bounded by a value dependent only on benign transactions. This definition reflects the following intuition: If the benign-transaction workload can provide enough block revenue for validators (i.e., the mempool end state without attack has enough benign transactions), the g-security of mempool ensures the mempool under the same benign workload and any adversarial workloads (i.e., attacks) has enough fees or enough block revenue for validators.

Definition 4.4 (Mempool locking security). Consider any timeline under attacks in which the mempool of initial state storing only benign transactions $\langle st_{0b}, \varnothing \rangle$ receives an ordered-transaction sequence interleaving adversarial and benign transactions, $\vec{as} = \vec{as_a} \oplus \vec{as_b}$, transitions its state, and reaches the end state $\langle st_n, dc_n \rangle$, as shown in Equation 5.

The mempool is considered 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,

$$\forall st_{0b}, \vec{as_b}, \vec{as_a}, \vec{as} = as_a \oplus as_b,$$
$$f(\langle st_{0b}, \varnothing \rangle, \vec{as}) \Rightarrow \langle st_n, dc_n \rangle \quad (5)$$
$$\exists \text{function } h(\cdot) < avgprice(\cdot) \quad (6)$$
$$\text{s.t., } maxprice(dc_n) < h(st_n)$$

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

4.1. Analysis of the Definitions

This subsection presents an informal analysis of the soundness and completeness of the proposed definitions. We do so from two angles: 1) how known attacks violate the definitions and 2) how all possible attacks, including the unknown ones, must violate the definitions.

Violation captures known attacks: In existing literature, 1) DETER attacks [25] follow one specified pattern (direct eviction), that is, sending invalid transactions to fully evict the valid transactions in the mempool. Clearly, it violates the g-eviction security where function $g()$ is simply a non-zero constant, say 1 wei. For a primary mempool pool storing only pending transactions, MemPurge [32] follows the same pattern as DETER by sending invalid transactions to evict valid transactions in the mempool.

2) MPFUZZ [31] is a fuzzer based on a more general insecurity definition, or bug oracle, which, however, is still very strict. The eviction bug oracle in MPFUZZ entails the non-overlapping between initial and end mempool states; the g-eviction security definition in this work removes such a condition. Thus, the different eviction attacks found in MPFUZZ are (true) positives in the negation of g-eviction security. For instance, given an MPFUZZ-compatible eviction attack with parameter $\epsilon = \frac{\sum_{st_n} tx.fee}{\sum_{st_0} tx.fee}$, one can easily find a price function $g(st_{nb}) = \epsilon \cdot \sum_{st_0} tx.price$ s.t. Inequality 4 is violated. A similar analysis applies to the locking bug oracle in MPFUZZ and h-locking security.

Violation captures all attacks: Conceptually, all asymmetric mempool DoSes succeed by exploiting cost asymmetry conditions; that is, the damage measured by the (normal) transactions eventually excluded from the mempool is significantly higher than the attack cost bounded by the transactions included in the blockchain. Our security definitions realize this cost asymmetry by mandating the lower bound of damage (by considering all excluded transactions are normal transactions) is higher than the upper bound of the attack cost (by considering all included transactions are adversarial transactions).

Specifically, our security definitions concretely capture two abstract attack patterns, i.e., eviction- and locking-attacks. We claim that the two abstract patterns are complete due to the reasons below: The damage of any asymmetric mempool DoS comes from the exclusion of normal transactions from a mempool's end state, which can come about in three possible ways: 1) declining an arriving normal transaction, 2) admitting the normal transaction only to evict it later, and 3) admitting the normal transaction only to turn it into an invalid one later.

Our security definitions capture all three possible damages: The eviction security quantifies the cost asymmetry in Cases 2) and 3), and the locking security quantifies the cost asymmetry in Case 1). If a given timeline is cast into an eviction-only (or locking-only) attack, the g-eviction definition (h-locking definition) is sufficient to ensure security in the worst-case scenario. Suppose the given timeline is cast into an interleaving of eviction and locking sub-attacks. What ensures the security is a conjunctive form of the two definitions, that is, Inequality 4 holds *and* Inequality 6 holds. Intuitively, the conjunctive form ensures that the total attack cost is higher than each sub-attack's upper bound of damage.

5. The SAFERAD Defense Framework

5.1. Design Rationale

Multi-pool framework: The SAFERAD mempool contains multiple sub-pools, out of which a primary sub-pool stores

only valid, pending transactions, and other secondary sub-pools may store invalid transactions temporarily.

Blocks are built by selecting transactions *only* from the primary sub-pool. Given an arriving transaction, the primary sub-pool attempts to admit it before other secondary sub-pools. Only valid transactions are admitted to the primary sub-pool. Invalid transactions, such as future transactions or overdrafts, are declined and may be admitted to secondary sub-pools. The invalid transactions buffered in the secondary sub-pool may attempt to enter the primary sub-pool.

Because the primary sub-pool controls what block builders (or validators) can see and has priority in admitting transactions, it is security critical. This work focuses on the secure design of the primary sub-pool.[†] We discuss revenue optimization in secondary sub-pools in § 10. Because other sub-pools buffering invalid transactions don't directly feed transactions to the block validator, their "security" under DoS is secondary.

Note that in our framework, the storage of valid transactions (in the primary sub-pool) is isolated from that of invalid transactions; the existing DETER attack that exploits the mixed storage of valid and invalid transactions becomes inapplicable and ineffective.

Design choices: A common mempool-design paradigm is to assign each transaction a score and use the score as the priority to decide transaction admission: Transactions of higher priority remain in the mempool, while those of lower priority are left outside (i.e., evicted from the mempool or declined by the mempool upfront). Admission scores are based on static transaction attributes (i.e., attributes independent of the context of a transaction or smart-contract execution), which makes the admission lightweight (e.g., without executing smart contracts) and free of resource-exhaustion risks [32]. To the best of our knowledge, static transaction scoring is adopted in the mempools of all Ethereum clients [5], [7], [3], [6], [8], [4], [2], [1].

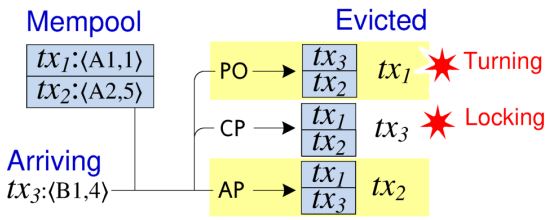


Figure 2: Motivating example: A two-slot mempool storing tx_1 and tx_2 receives an arriving transaction tx_3.

In practice, mempools use transaction scores such as transaction price, transaction age (i.e., the order transactions arrive at the mempool), etc. However, these scores don't take into account transaction validity and can be misused. For instance, a mempool that admits transactions only by price (i.e., the *price-only* score or **Score PO**) can be misused to trigger dangerous eviction behavior. Here, we use an example depicted in Figure 2 where a two-slot mempool initially stores transactions $tx1 = \langle A1, 1\rangle, tx2 = \langle A2, 5\rangle$ and receives an arriving transaction $tx_3 = \langle B1, 5\rangle$. If the mempool exercises Policy PO, it would admit tx_3 and select from the pool the transaction of the lowest price to evict, that is, tx_1. The eviction would turn tx_2 into a future transaction, rendering its fee unchargeable and the mempool transitioning from a high-cost state into a low-cost one (i.e., from $1+5$ to $4+0$). Transaction turning has been misused in constructing eviction-based mempool DoS, such as the XT_6 discovered by MPFUZZ on Geth (see § 8.3).

To avoid transaction turning, the key is to prevent evicting "parental" transactions that have any descendants in the mempool. We propose admission schemes that enforce the following invariant: A parent transaction is always assigned with a higher score than any of its descendants. We employ an admission algorithm that selects transactions of the lowest score as eviction victims so that the selected transaction has no descendants to turn (the properties that will be proven in § 6 as Lemmas 6.3 and 6.5).

5.2. The Framework and Policies

Recall that Algorithm ADTX(st, ta) determines whether to admit an arriving transaction ta at the mempool of state st. To realize such an algorithm, the key idea in this work is to treat the mempool as an *ordered* list of transactions and to use such transaction ordering as the priority in admission to achieve security. Specifically, we propose the concept of transaction *scores* by which transactions are ordered; our admission algorithm ADTX() ensures that transactions admitted into the mempool always have higher scores than those left outside.

This section describes the two transaction scores and the proposed admission algorithms.

Score CP (Childless Price): We propose the first scoring function CP where any "parental" transaction that has at least one descendant transaction in st is assigned with an infinitely large score ($+\infty$), and any "childless" transaction in st is assigned a score equal to its price. That is,

$$score_{\text{CP}}(tx, st) = \begin{cases} tx.price, \text{ if } tx.descendants() \notin st \\ +\infty, \text{ otherwise.} \end{cases}$$

Score AP (Ancestor Min. Price): We first define the minimal-price ancestor: Given a transaction in a set, $tx \in st$, the minimal-price ancestor, denoted by $txm(tx, st)$, is the transaction whose price is the lowest among all ancestors of tx in st.

Given a mempool's (unordered) state st, transaction tx's ancestor minimal price or AP equals $txm(tx, st)$'s price:

$$txm(tx, st).price = \min_{\substack{\forall tx' \in st \cap \\ tx.ancestors()}} tx'.price \quad (7)$$

$$score_{\text{AP}}(tx, st) = txm(tx, st).price \quad (8)$$

[†]In the rest of the paper, unless otherwise specified, we use the words "mempool" and "primary sub-pool" interchangeably.

Example: Consider the example scenario described in Figure 2. Specifically, a mempool of state st initially stores two transactions, $tx_1 = \langle A1, 1\rangle$ and $tx_2 = \langle A2, 5\rangle$. That is, both transactions are sent from Alice, and they are, respectively, with nonces 1 and 2, and of prices 1 and 5. A transaction $tx_3 = \langle B1, 4\rangle$ arrives at the mempool.

We have the following scores: $score_{\text{CP}}(tx_1, st) = \infty$, $score_{\text{CP}}(tx_2, st) = 5$, $score_{\text{CP}}(tx_3, st) = 4$, $score_{\text{AP}}(tx_1, st) = 1$, $score_{\text{AP}}(tx_2, st) = 1$, and $score_{\text{AP}}(tx_3, st) = 4$.

It is clear from Equation 8 that given any transaction tx and set st, $score_{\text{AP}}(tx, st)$ is not higher than $tx.price$. This property is useful in proving the eviction security, as will be seen. Formally,

$$\forall tx, st, \text{ s.t. } tx \in st$$
$$score_{\text{AP}}(tx, st) \leq tx.price \tag{9}$$

Admission Algo. AA: 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, $te = ta$.

Algorithm 1 ADTX$_{AA}$(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: st⃗=TORDER(st, score(·));
5: te=st⃗.lastTx();
6: if isFull(st) then
7:     if score(ta, st) ≤ score(te, st) 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
```

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 proceeds if ta passes the validity pre-check.

It then sorts all transactions in mempool st in descending order based on a selected scoring function, $score()$ (in Line 4). As described next, function $\vec{st} = \text{TORDER}(st)$ produces a total order of transactions in the mempool, denoted by $\vec{st}$. It selects the last transaction on this ordered list, denoted by te. That is, te has the lowest score on $\vec{st}$.

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

One can plug different $score()$ functions, including CP and AP, into Algorithm 1 to obtain different admission policies. Thus, we also use the $score$ function name, such as AP, to refer to the corresponding admission policy.

Tx total-order by TORDER(): To produce a total order among transactions, we enforce the following rule: Given two transactions, tx_1 and tx_2, if $score(tx_1, st) < score(tx_2, st)$, tx_1 is ordered after tx_2. If $score(tx_1, st) = score(tx_2, st)$, it breaks even in two cases: 1) If $tx_1.sender = tx_2.sender$, it orders the two transactions by their nonces: If $tx_1.nonce > tx_2.nonce$, tx_1 is ordered after tx_2. 2) If $tx_1.sender \neq tx_2.sender$, it uses the following heuristics: Given a deterministic hash function $H(\cdot)$, if $H(tx_1.sender) < H(tx_2.sender)$, tx_1 is ordered after tx_2.

TABLE 2: Admission policies and their security. OI means order insensitivity as in Definition 6.1.

Policies	Eviction security		Locking security
	Without OI	With OI	
CP	✓	✗	✗
AP	✓	✓	✓

6. Security Analysis

Before we analyze the security of individual policies, we establish a security-proof framework based on the property definitions in § 6.1.

6.1. Eviction-Security Definitions

Given an ordered list of transactions $\vec{as}$, a reordered list of transactions $\vec{as'}$ is considered to preserve the parent-before relationship, if.f. for any $\forall tx_i, tx_j \in \vec{as}$ such that tx_i is a parent of tx_j and tx_i is ordered before tx_j on $\vec{as}$ (i.e., Equation 12), tx_i is ordered before tx_j on $\vec{as'}$ (Equation 11).

Definition 6.1 (Order insensitivity). A mempool is admission-order insensitive if.f. given any timeline $\vec{as}$ (in Equation 10) and any reordered timeline preserving the parent-before relationship $\vec{as''}$, the mempool always reaches the same end state, $\langle st_n, dc_n\rangle \equiv \langle st''_n, dc''_n\rangle$. Formally,

$$\forall f(\langle st_{0b}, \varnothing\rangle, \vec{as}) \Rightarrow \langle st_n, dc_n\rangle \tag{10}$$
$$\forall f(\langle st_{0b}, \varnothing\rangle, \vec{as''}) \Rightarrow \langle st''_n, dc''_n\rangle \text{ s.t.}$$
$$unorder(\vec{as}) = unorder(\vec{as''}) \wedge$$
$$\forall tx_i, tx_j \in \vec{as}, i < j \wedge \tag{11}$$
$$tx_i \in tx_j.ancestors() \wedge$$
$$\vec{as''}.idx(tx_i) < \vec{as''}.idx(tx_j) \tag{12}$$
$$\langle st''_n, dc''_n\rangle \equiv \langle st_n, dc_n\rangle$$

Definition 6.2 (Monotonic score-increasing). A mempool admitting transactions is monotonically score-increasing, if.f. under an arbitrary timeline $f(\langle st_0, \varnothing\rangle, as) = \langle st_n, dc_n\rangle$, the total $score$'s of all the transactions in the mempool monotonically increases. Formally,

$$\forall f(\langle st_0, \varnothing\rangle, as) = \langle st_n, dc_n\rangle, \quad (13)$$
$$\sum_{tx' \in st_n} score(tx, st_n) \geq \sum_{tx \in st_0} score(tx, st_0)$$

6.2. Security of Policy CP

Eviction security (violating OI): Policy CP achieves weak eviction security in the sense that it ensures the total prices monotonically increase, but it breaks order insensitivity.

Lemma 6.3 (No tx turning of* CP*). Suppose a mempool runs Algorithm 1 under $score_{\text{CP}}$ 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}.

Theorem 6.4 (Monotonic price-increasing). If a mempool runs Algorithm 1 under $score_{\text{CP}}$, the sum of transaction prices in the mempool monotonically increases, or the mempool is considered to be monotonic price-increasing.

Due to the space limit, the proofs of Theorem 6.4 and Lemma 6.3 are deferred to the technical report [20].

However, Policy CP breaks order-insensitivity. Consider a simple counter-example that extends the case in Figure 2 with another transaction tx_4 to arrive after tx_3. $tx_4 = \langle C1, 6\rangle$. In the original timeline when tx_3 arrives before tx_4, the mempool running Policy CP declines tx_3 (because $score_{\text{CP}}(tx_3, st_0) = 4 < 5 = score_{\text{CP}}(tx_2, st_0)$) and admits tx_4 by evicting tx_2, leaving the mempool at end state st_2 storing tx_1 and tx_4.

On the reordered time when tx_4 arrives before tx_3, the mempool running Policy CP admits tx_4 by evicting tx_2, leaving the mempool at intermediate state st'_1 of tx_1, tx_4. The next transaction to arrive, tx_3, would be admitted into the mempool at this intermediate state, because $score_{\text{CP}}(tx_3, st'_1) = 4 > 1 = score_{\text{CP}}(tx_1, st'_1)$. The mempool end state st'_2 stores tx_3 and tx_4. Thus, $st'_2 \neq st_2$, that is, the two timelines reach different mempool end states.

Locking insecurity: Policy CP is not locking secure. Figure 2 shows a counterexample in which we can easily construct an effective locking attack. Specifically, recall that in the figure, tx_3 is declined by the mempool running Policy CP (i.e., Algorithm 1 with $score_{\text{CP}}$). Assume transactions tx_1 and tx_2 are sent by an adversary (i.e., Adversarial account A), and tx_3 is sent by a benign user (B). This admission event implies that a benign user's transaction of price 4 is declined by a mempool where the average transaction price per slot is $\frac{1+5}{2} = 3$, which is lower than 4. Suppose there is another subsequent normal transaction $tx_5 = \langle D1, 4\rangle$. tx_5 would be declined by the mempool. Had tx_3 and tx_5 been admitted into the mempool, the sum of prices in the end state would have been $4 + 4 = 8$, which is higher than the sum of prices in the actual mempool end state, $1 + 5 = 6$. This is an asymmetric mempool-locking attack. The example also violates h-locking security, where h must be lower than the average transaction price.

6.3. Security of Policy AP

Eviction security: We first analyze the eviction security of Policy AP.

Lemma 6.5 (No tx turning of* AP*). Suppose a mempool runs Algorithm 1 under $score_{\text{AP}}$ 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}.

The proof of Lemma 6.5 is in Appendix B.

Theorem 6.6 (monotonic $score_{\text{AP}}$-increasing). A mempool running Algorithm 1 under $score_{\text{AP}}$ is monotonic $score_{\text{AP}}$-increasing as in Definition 6.2.

The proof of Theorem 6.6 is deferred to Appendix B.

Theorem 6.7 (Order-insensitivity of* AP*). A mempool running Algorithm 1 under $score_{\text{AP}}$ is admission-order insensitive as in Definition 6.1.

Proof sketch: Algorithm 1 sorts transactions in the mempool by $score_{\text{AP}}$ and maintains such ordering across arbitrary admission events. Because a transaction's $score_{\text{AP}}$ is static, running Algorithm 1 with Policy AP is order insensitive: Given the same (unordered) set of transactions arriving, no matter how they are ordered, the ordered list of transactions inside the mempool end state reached by Policy AP is always the same. The full proof is in Appendix § A.

Theorem 6.8 (g-eviction security of* AP*). A mempool running Algorithm 1 under $score_{\text{AP}}$ is g-eviction secure. Given any timeline without attacks (Equations 3) and any timeline with attacks (Equation 2), the total transaction fees of the mempool end-state under attacks (st_n) are lower-bounded by the following, no matter how the transactions in st_n are ordered when building blocks:

$$\begin{aligned} \forall \vec{st_n}, fees(\vec{st_n}) &\geq g_{\text{AP}}(st_{nb}) \quad (14)\\ &= 21000 \cdot \sum_{tx' \in st_{nb}} score_{\text{AP}}(tx', st_{nb}) \end{aligned}$$

Proof Due to the security definition 4.3, we prove the theorem by considering any timeline under attacks, that is, Equation 2.

We construct the following timeline, where benign transactions and adversarial transactions are not interleaved:

$$\begin{aligned} f(\langle st_{0b}, \varnothing\rangle, a\vec{s}_b | a\vec{s}_a) &\Rightarrow \langle st'_n, dc'_n\rangle\\ \text{Specifically, } f(\langle st_{0b}, \varnothing\rangle, a\vec{s}_b) &\Rightarrow \langle st_{nb}, dc_{nb}\rangle\\ f(\langle st_{nb}, dc_{nb}\rangle, a\vec{s}_a) &\Rightarrow \langle st'_n, dc'_n\rangle \end{aligned}$$

The timeline is illustrated as in Figure 3. Because $unorder(\vec{as}) = unorder(a\vec{s}_a \oplus a\vec{s}_b) = unorder(a\vec{s}_b | a\vec{s}_a)$

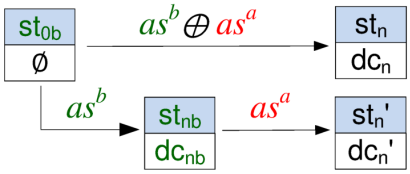


Figure 3: Reorder the timeline to have separated benign and adversarial transaction lists for proving eviction security. Green are benign transactions and red are adversarial transactions.

and due to order insensitivity in Theorem 6.7, the newly constructed timeline reaches the same end state as the original timeline. Thus,

$$\langle st'_n, dc'_n \rangle = \langle st_n, dc_n \rangle$$

Therefore, we can have a timeline: $f(\langle st_{nb}, dc_{nb} \rangle, a\vec{s}_a) \Rightarrow \langle st_n, dc_n \rangle$. Applying Theorem 6.6 (monotonic increasing of $score_{\text{AP}}$) to this timeline leads to the following:

$$\sum_{tx' \in st_n} score_{\text{AP}}(tx', st_n) \geq \sum_{tx' \in st_{nb}} score_{\text{AP}}(tx', st_{nb})$$

Due to Equation 9, we derive:

$$\begin{aligned} &\therefore \forall \vec{st_n} = \text{TORDER}(st_n), \\ &fees(\vec{st_n}) \geq 21000 \cdot \textstyle\sum_{tx' \in st_n} tx.price \\ &\geq 21000 \cdot \textstyle\sum_{tx' \in st_n} score_{\text{AP}}(tx', st_n) \\ &\geq 21000 \cdot \textstyle\sum_{tx' \in st_{nb}} score_{\text{AP}}(tx', st_{nb}) \end{aligned}$$

Equation 14 holds.

Example: Consider Figure 2 that depicts the admission of arriving tx_3 at the mempool of state: tx_1 and tx_2. Assume tx_1 and tx_2 are benign transactions, and tx_3 is adversarial; that is, $st_{nb} = st_0$ consists of tx_1 and tx_2.The lower fee bound from Equation 14 is $21000 \cdot \sum_{tx' \in st_{nb}} score_{\text{AP}}(tx', st_{nb}) = 21000 * \cdot (1+1) = 42000$.

Locking security: We then analyze the locking security of Policy AP.

Lemma 6.9 (Monotonic increasing $\min score_{AP}$***).*** For any timeline of a mempool running Algorithm 1 with Policy AP, denoted by $f(\langle st_0, \varnothing \rangle, \vec{as}) \Rightarrow \langle st_n, dc_n \rangle$, the following holds:

$$\begin{aligned} \forall 0 \leq i < j \leq n, \min_{tx \in st_i} score_{\text{AP}}(tx, st_i) \\ \leq \min_{tx \in st_j} score_{\text{AP}}(tx, st_j) \end{aligned} \quad (15)$$

Proof of Lemma 6.9 is in Appendix § C.

Theorem 6.10 (h***-locking security of* AP*).*** A mempool running Algorithm 1 with Policy AP is h-locking secure, with $h(st) = (1+\gamma) \min_{tx' \in st_n} score_{\text{AP}}(tx', st_n)$ with $\gamma \geq 0$, under the following assumption.
Given any benign transaction tx_C, for any set of transactions that are ancestors to tx and whose nonces are consecutive w.r.t. tx, say ts, $tx_C.price$ must be lower than tx_C's minimal ancestor price in ts multiplied by $1+\gamma$. Formally,

$$\begin{aligned} &\forall tx_C, \forall ts, \text{ISFUTURE}(tx_C, ts) = 0 \\ &tx_C.price < (1+\gamma) score_{\text{AP}}(tx_C, ts) \end{aligned} \quad (16)$$

Due to the space limit, the proof of Theorem 6.10 is deferred to our technical report [20].

7. Implementation Notes on Geth

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

Background: Geth mempool implementation: In Geth $v1.11.4$, the mempool adopts the price-only Policy (PO) 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 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 [16]: A MemPurge attack works by reconnecting future transactions only to turn them into overdraft transactions. The attack is mitigated by the patch which checks and invalidates overdraft transactions when reconnecting future transactions.

Implementation of SAFERAD 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 transaction with the lowest $score()$ in the mempool. If $score_{\text{CP}}$ is used, in Step ①, we 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 descendant 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.

If $score_{\text{AP}}$ is used, we build another index that keeps track of the ancestor transaction with the lowest price for each transaction. Upon the admission or eviction of a transaction in the mempool, the index does not need to be updated because transaction admission by Algorithm 1 does

not change an existing transaction's ancestors and thus its $score_{\text{AP}}$. Upon transaction replacement, the index needs to be updated, and the transaction entries with the same sender to the replacement transaction need be re-calculated.

Steps ②, ③, and ④ remain the same except that reference te' is replaced with te.

8. Evaluation of Block Revenue

This section evaluates the block revenue of different SAFERAD policies under normal and attacks workloads.

8.1. 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 were 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 senders' secret keys and can send the otherwise same transactions in the experiments.

We choose 8 traces of consecutive transactions from the raw dataset collected, each lasting 2.5 hours. 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.

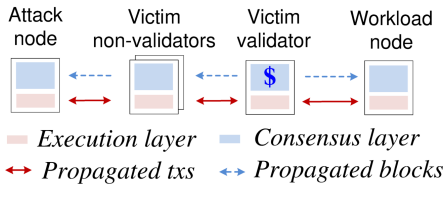


Figure 4: Experimental setup

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 run the target Ethereum client to be tested; we tested three victim clients: vanilla Geth $v1.11.4$, SAFERAD CP and SAFERAD AP; the latter two are implemented as add-don 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 4. 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 under attacks, that is, how successful DoS attacks are on SAFERAD. In this setting, we run all three nodes (the victim, attack and workload nodes). For the latter, we aim to evaluate the utility of SAFERAD 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. 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.

Method limitation: Our transaction-replay method presents a coarse estimation of revenue due to the following reasoning: First, our method can precisely reproduce the same transaction fees if the same blocks (and transaction ordering) are produced in our experiment as in the mainnet; because we use $GasUsed$ in replaying those transactions included in the mainnet. Second, our method does not cover the validators' revenue received from smart contracts (e.g., MEV and bribing).' Third, our method presents a coarse estimation of fees for transactions excluded from the mainnet blockchain. For those transactions, there is no ground truth regarding their $GasUsed$; the value of $GasUsed$ depends on runtime factors like transaction ordering and timing, which are unknown for transactions excluded from the blockchain. Generally, the possible values for $GasUsed$ fall in a range, and our method is a best-effort estimation of such a range based on the assumption that, in practice, transaction senders are incentivized to make their $tx.Gas$ as precise as possible to the expected range of $GasUsed$.

8.2. Revenue Under Normal Transactions

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

We consider three mempool policies, the baseline one in Geth $v1.11.4$, SAFERAD-CP and SAFERAD-AP 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.

Figure 5 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 3 presents the aggregated results by the revenue per block. Compared to Geth $v1.11.4$, CP's revenue per block falls in the range of $[100\%{+}1.18\%, 100\%{+}7.96\%]$. AP's revenue per block falls in the range of $[100\% - 1.33\%, 100\% + 2.55\%]$. This result suggests under CP or AP, SAFERAD incurs no significant change of block revenue under normal transactions.

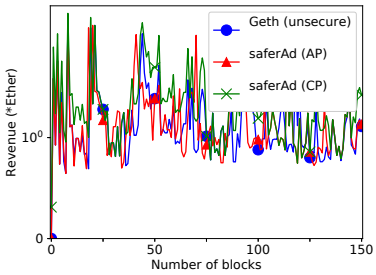


Figure 5: Block revenue w/w.o. SAFERAD under Trace 2. SAFERAD introduces negligible change of revenue.

TABLE 3: Block revenue (Ether) of different mempool policies. In bold are max and min numbers.

Trace	CP	AP	Geth
1	0.73 (+3.13%)	0.72 (+1.02%)	0.71
2	1.17 (+6.97%)	1.08 (−1.19%)	1.09
3	0.85 (+4.55%)	0.81 (−0.21%)	0.81
4	0.54 (+4.56%)	**0.54 (+2.55%)**	0.52
5	0.78 (+6.66%)	0.72 (−1.09%)	0.73
6	1.19 (+4.23%)	**1.12 (-1.33%)**	1.14
7	**0.56 (+1.18%)**	0.57 (+2.1%)	0.55
8	**0.64 (+7.96%)**	0.59 (+0.74%)	0.59

8.3. Revenue Under Known Eviction Attacks

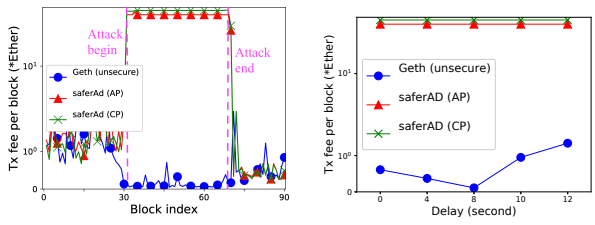
This experiment evaluates the security of SAFERAD against known eviction attacks.

Background of eviction attacks: Given that different mempools are exploitable to different eviction attacks, we choose a variant of eviction attack known as XT_6 [31] that is found to be successful on Geth $v1.11.4$. Briefly, XT_6 works on the mempool of Geth $v1.11.4$, whose capacity is of $m = 5120$ transactions. Geth's mempool can additionally store up to $m_f = 1024$ future transactions. The mempool admits up to $l = 16$ transactions of the same sender, provided there are less than 5120 pending transactions in the $m + m_f = 6144$ slots.

Under this setting, XT_6 works in the following four steps: 1) It first evicts the Geth mempool by sending $\frac{m+m_f}{l} = 384$ transaction sequences, each of $l = 16$ transactions from a distinct sender. The transaction fees are high enough to evict normal transactions initially in the mempool. 2) It then sends $\left\lceil \frac{m_f}{l-1} \right\rceil = 69$ 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 $m = 5120$ pending transactions in the mempool, Geth's limit of 16 transactions per sender 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 until it is with one transaction, we estimate Geth's bound by the maximal price in a given mempool times the minimal Gas per transaction (21000 Gas).

Experimental method: We set up the experiment platform described in § 8.1. 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 eviction attack. Each time the attack node detects the arrival of a newly produced block, it waits for d seconds and sends a round of crafted transactions. The attack phase lasts for 36 blocks; after the 66-th block, we stop the eviction attack node from sending crafted transactions. We keep running the workload and victim nodes for another 24 blocks, then stop the process at the 90-th block. We collect produced blocks and, for each block, report the fees of included transactions replayed using the method described in § 8.1.



(a) With 8-second delay (b) With varying delay

Figure 6: Validator revenue under attacks (XT_6 [31]): With and without SAFERAD defenses on a single node.

Results: Figure 6a reports the metrics for XT_6 on three victim clients: vanilla Geth $v1.11.4$, SAFERAD-AP and SAFERAD-CP. Before the attack is launched (i.e., before the 30-th block), the three 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-AP and SAFERAD-CP can

fix the vulnerability. After the attack starts on the 30-th block, the 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 SAFERAD's effectiveness in defense against known eviction-based attacks (XT_6).

Figure 6b shows the fees 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 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 Policy CP and AP would remain constant at a high value, showing the defense effectiveness against attacks of varying delays.

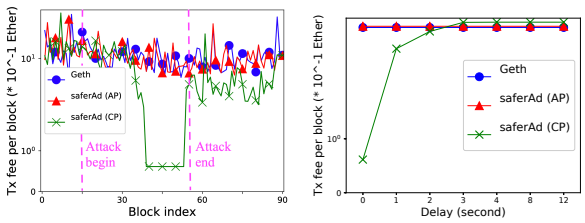
8.4. Revenue Under Known Locking Attacks

This experiment evaluates the security of SAFERAD against known locking attacks.

Background of locking attacks: Recall that in our Policy CP, an arriving transaction ta evicts a childless transaction te only when $ta.price > te.price$. In practice, OpenEthereum adopts this policy and is found exploitable under a locking attack named by XT_9 in MPFUZZ [31]. We use XT_9 as the attack workloads for evaluation. In XT_9, the attacker monitors for empty slots in the mempool, which occur due to the arrival of new blocks, and promptly sends the same number of transactions to fill these slots. These transactions are distributed across $\frac{m+m_f}{l} = 384$ different accounts. Each transaction set includes a child transaction with a higher *price* p_h compared to normal transactions, while all parent transactions have a minimal fee of 1 $Gwei$. As a result, subsequent normal transactions are declined because their *price* is lower than p_h. Since the parent transactions have a very low *price*, the mempool becomes locked at a low overall cost.

Experimental method: We run experiments using the same method as in § 8.3 except that the attack in this experiment begins at the 15-th block and ends at the 55-th block, lasting for 40 blocks.

Results: Figure 7a shows the metrics under XT_9 for three clients. Before the attack (up to the 15-th block), fees are similar (0.7 - 2.0 Ether per block). From the 15-th to the 30-th block, the attacker fills the mempool, maintaining this fee range due to sufficient normal transactions. After the 30-th block, SAFERAD-CP fees drop to 0.06 Ether per block as the mempool is filled with the attacker's low-fee parent transactions and a single high-fee childless transaction, blocking normal transactions. Even after the attack ends at the 55-th block, block revenue remains low due to lingering low-fee attack transactions. This shows XT_9 effectively locks



(a) XT_9 w. 0-sec. delay (single node)

(b) XT_9 w. varying delay (single node)

Figure 7: Validator revenue under attacks: With and without SAFERAD defenses.

the mempool under SAFERAD-CP, negatively impacting block revenue. In contrast, SAFERAD-AP and vanilla Geth maintain normal block revenue, demonstrating SAFERAD-AP's effectiveness against XT_9.

Figure 7b shows the fees per block under XT_9 with varying delays. For SAFERAD-CP, a short delay results in low transaction fees due to the rapid locking of the mempool by XT_9. Low-revenue blocks emerge early, resulting in the reduced fee per block. Conversely, with a long delay, transaction fees increase as normal transactions occupy empty slots and are mined into new blocks, thereby raising the average revenue of blocks. SAFERAD-AP and vanilla Geth maintain consistently high fees regardless of delay, demonstrating their effectiveness against locking attacks.

8.5. Estimation of Eviction Bounds

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

Experimental method: In the experiment, we replayed the transaction traces we collected in § 8.1 on one of three Ethereum clients, be it either CP, AP, 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 the sum of price in st_i to estimate the above bound, that is, $21000 * \sum_{tx \in st} tx.price$. 2) For AP, we use $g_{\text{AP}}(st_{i+10} \cup bk_{i+1} \cup bk_{i+2} \ldots \cup bk_{i+10})$ to estimate the above bound; recall $g_{\text{AP}}()$ in Equation 14. 3) 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.

Results: Figure 8 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, and the bound of AP is higher than that of Geth $v1.11.4$ by 5 orders of magnitude.

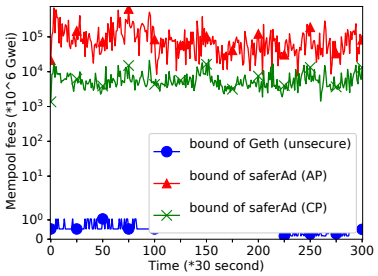


Figure 8: The lower fee bounds of CP, AP, and vanilla Geth $v1.11.4$ under worst-case eviction attacks.

TABLE 4: 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
AP	101.05	299.60	16.41
Geth	$0.63 \cdot 10^{-3}$	$1.05 \cdot 10^{-3}$	$0.17 \cdot 10^{-3}$

Table 4 presents statistics of the estimated bounds on the three clients tested. It includes the average, 95-th, and 5-th percentile of eviction bounds. Both SAFERAD policies achieve statistically higher bounds than the baseline of Geth. On average, CP's eviction bound is 6.05 Ether, and AP's bound is 101.05 Ether, both of which are much higher than the $0.63 \cdot 10^{-3}$ Ether in Geth under attacks. For CP, 5% of its bounds exceed 13.72 Ether, and 95% exceed 2.63 Ether. For AP, 5% of its bounds exceed 299.60 Ether, and 95% of its bounds exceed 16.41 Ether.

8.6. Estimation of Locking Bounds

This experiment estimates the γ value in Equation 16 which indicates the security level of the mempool under locking attacks. Since Policy CP is unsecured against locking, we focus on evaluating AP here. Given a sequence of arriving transactions $o\vec{p}s$, we derive from Equation 16 the following: $\gamma(tx) = \frac{tx.price}{score_{AP}(tx, o\vec{p}s)} - 1$, $\gamma(s) = \max_{\substack{\forall tx \in o\vec{p}s \wedge \\ tx.sender=s}} \gamma(tx)$, and $\gamma = \max_{\forall tx \in o\vec{p}s} \gamma(tx)$.

We similarly run experiments as in § 8.5 with the eight collected transaction traces described in § 8.1.

TABLE 5: Estimated γ from eight transaction traces.

Tx trace	1	2	3	4	5	6	7	8
γ	0.82	0.37	0.33	0.24	0.50	0.37	**0.92**	0.35

The results are illustrated in Table 5. The maximal γ is 0.92 across the eight traces. This implies that under arbitrary attacks, the maximal price of declined transactions by a mempool running Algorithm 1 can be bounded by $1.92\times$ of the minimal price of transactions in the mempool.

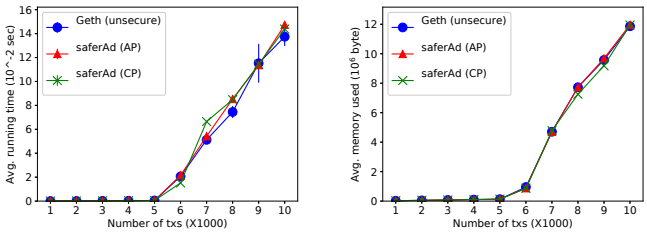
9. Performance Evaluation

9.1. Performance under Generic Workloads

Design rationale: This section evaluates the concrete performance overhead introduced by the SAFERAD defenses. As described by the implementation notes in § 7, some scoring functions entail additional index whose overhead is data dependent and can be superlinear (e.g., one transaction replacement on the index built for $score_{\text{AP}}$ can potentially trigger the entire scan of the mempool), we present an evaluation of performance overhead of SAFERAD under real-world transaction workloads.

Platform setup: In this work, we use Ethereum Foundation's test framework [12] 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 the provided workload (i.e., "Batch insert").

Results under "Batch insert" workload: 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 transactions 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 reorganizes the mempool (i.e., Function `pool.scheduleReorgLoop()` in Geth).



(a) Running time (b) Memory usage

Figure 9: Performance under Workload "Batch insert".

We report running time and memory usage in Figure 9a and Figure 9b, 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 in 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

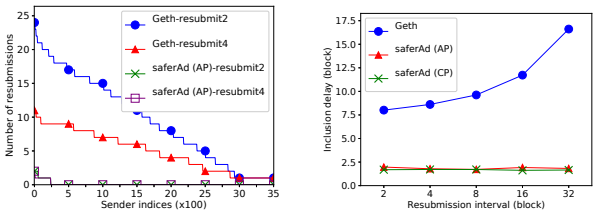
mempool ($n_0 > 5000$), on average, Geth $v1.11.4$ hardened by SAFERAD CP/AP causes $1.073\times$/$1.095\times$ running time and $1.000\times$/$0.994\times$ memory usage, compared to vanilla Geth $v1.11.4$. 2) When the mempool is not full ($n_0 \leq 5000$), on average, Geth $v1.11.4$ hardened by SAFERAD CP/AP causes $1.017\times$/$1.005\times$ running time and $1.000\times$/$1.000\times$ memory usage, compared to vanilla Geth $v1.11.4$. Overall, the overhead by SAFERAD-CP/AP 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.

9.2. Performance under Tx Resubmission

Methods: In practice, if a sender experiences a long delay in block inclusion or dropped transactions, they will commonly retry sending the transaction to ensure the block inclusion. We evaluate SAFERAD under this adaptive user behavior.

Specifically, we aim to report and study two metrics: the number of resubmitted transactions and the inclusion delay (i.e., the time between when the transaction is first submitted and when the transaction is eventually included in a block (or when the attack ends)). To do so, we implement a basic transaction-resubmitting strategy, that is, the user waits a fixed period if she does not see her previously sent transaction included in the blockchain. There are many other "backoff" strategies [15], the modeling and evaluation of which are out of the scope. We vary the time interval between transaction re-submission, and for each interval, we run the experiment under attacks in the same way as in § 8.3 and § 8.4.



(a) Distribution of resubmission times over senders

(b) Avg. inclusion delay (under eviction attacks)

Figure 10: SAFERAD and vanilla Geth with transaction resubmission

Results are presented in Figures 10a and 10b. Given each experimental run, we rank the transaction senders by the number of times their transactions are resubmitted. We plot the distribution of resubmission times over senders in Figure 10a. It is clear that SAFERAD entails much fewer resubmissions than the vanilla Geth without protection. For Geth, the shorter the interval is, the more resubmissions are.

Figure 10b presents the average inclusion delay with varying resubmission intervals. This experiment carries out eviction attacks. It is clear that SAFERAD achieves much shorter inclusion delay (or transaction finality delay) than unprotected Geth under user resubmission behavior. With the increasing intervals, the delay also gets increased as reasoned below: Given that a Geth node is vulnerable under eviction attacks but not locking attacks, a resubmitted transaction can be included into a block if it occurs after an eviction attack but before the validator builds the next block. The shorter the resubmission interval is, the more likely the resubmitted transaction can hit this time window and be included.

10. Discussion

Handling invalid transactions: This work focuses on the secure design of the primary mempool (i.e., the one storing only pending transactions) and considers the "basic" setting where the primary mempool is of fixed size.

In practice, however, one can optimize mempool utility or specifically block revenue by adding secondary mempools to temporarily buffer invalid transactions, such as future transactions and overdrafts. Specifically, there are three cases: 1) An arriving invalid transaction declined by the primary pool (recall the pre-check Line 1 in Algorithm 1) enters the corresponding secondary pool. 2) An invalid transaction buffered in the secondary pool may be "promoted" as a valid transaction (e.g., a future transaction in the secondary pool whose parent transaction recently arrived). The promotion triggers the transaction to re-enter the primary mempool. 3) A transaction evicted from the primary pool will not enter any secondary pool.

In the above design, there is no interference between the primary pool and secondary pool, in the sense that no invalid transaction can cause the eviction of an existing valid transaction in the primary mempool (in compliance with the attack-specific defense proposed in DETER [25] and MPFUZZ [31]).

However, this non-interference design can lead to sub-optimal block revenue: When an invalid transaction arrives at a full secondary pool and a primary pool with empty slots, the transaction would be declined by both pools in the current design. One can further optimize block revenue by repurposing empty slots in the primary pool to store invalid transactions. When this occurs, the primary pool treats an empty slot, storing an invalid transaction as if it were a normal empty slot; that is, a slot buffering an invalid transaction in the primary pool would have the same priority in transaction admission as an empty slot.

Handling transaction replacements: SAFERAD handles transaction replacements in the same way as existing works [25], [31]. An imminent replacement of an existing transaction tx_2 by an arriving transaction tx_1 is checked to avoid turning any of tx_1's descendants into latent overdrafts.

Design generalizability: Our mempool security definitions and defenses are based on some generic transaction attributes and can apply to other major blockchains, notably Bitcoin. Specifically, this work only assumes transactions can form a parent-child relationship, such as in Algorithm 1 and scores like CP; such relationship models the case in Ethereum (where the parent and children are based on nonce) and the case in Bitcoin (where the parent and children are based on transaction input and output). Thus, given that Bitcoin transactions also have other attributes assumed

in this work, such as price, we believe it is straightforward to apply the techniques in this work to Bitcoin.

11. Conclusion

This paper presents provably secure mempool designs under asymmetric DoS attacks. It formulates security definitions under two abstract DoS attack types, namely eviction- and locking-based attacks. It presents a suite of secure mempool policies, SAFERAD, that achieves both eviction- and locking- security. The evaluation shows SAFERAD incurs negligible overhead in latency and validator revenue.

Acknowledgments

The authors thank the IEEE S&P 2025 reviewers for their constructive comments. This work was partially supported by two Ethereum Foundation academic grants and NSF grants CNS-2139801, CNS-1815814, DGE-2104532.

References

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

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

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

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

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

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

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

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

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

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

[11] Txpool validation rules have been tightened to defend against certain dos attacks. https://github.com/ethereum/go-ethereum/releases/tag/v1.11.4, Retrieved July, 2023.

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

[13] Defense against mempurge attack by checking for overdraft when adding tx. https://github.com/ethereum/go-ethereum/pull/28076, Retrieved May, 2024.

[14] 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.

[15] Exponential backoff. https://en.wikipedia.org/wiki/Exponential_backoff, Retrieved Oct, 2024.

[16] 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.

[17] M. Apostolaki, A. Zohar, and L. Vanbever. Hijacking bitcoin: Routing attacks on cryptocurrencies. In IEEE Symposium on SP 2017, pages 375–392, 2017.

[18] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver. Stressing out: Bitcoin "stress testing". In J. Clark, S. Meiklejohn, P. Y. A. Ryan, D. S. Wallach, M. Brenner, and K. Rohloff, editors, Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, volume 9604 of Lecture Notes in Computer Science, pages 3–18. Springer, 2016.

[19] V. Buterin. Eip150: Gas cost changes for io-heavy operations.

[20] W. Ding, Y. Wang, Z. Yang, and Y. Tang. Asymmetric Mempool DoS Security: Formal Definitions and Provable Secure Designs. https://arxiv.org/abs/2407.03543, 2024.

[21] Z. He, Z. Li, A. Qiao, X. Luo, X. Zhang, T. Chen, S. Song, D. Liu, and W. Niu. Nurgle: Exacerbating resource consumption in blockchain state storage via mpt manipulation. In 2024 IEEE Symposium on Security and Privacy (SP), pages 128–128, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[22] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In J. Jung and T. Holz, editors, USENIX Security 2015, Washington, D.C., USA, pages 129–144. USENIX Association, 2015.

[23] K. Li, J. Chen, X. Liu, Y. R. Tang, X. Wang, and X. Luo. As strong as its weakest link: How to break blockchain dapps at RPC service. In 28th Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021. The Internet Society, 2021.

[24] K. Li, Y. Tang, J. Chen, Y. Wang, and X. Liu. Toposhot: uncovering ethereum's network topology leveraging replacement transactions. In D. Levin, A. Mislove, J. Amann, and M. Luckie, editors, IMC '21: ACM Internet Measurement Conference, Virtual Event, November 2-4, 2021, pages 302–319. ACM, 2021.

[25] K. Li, Y. Wang, and Y. Tang. DETER: denial of ethereum txpool services. In Y. Kim, J. Kim, G. Vigna, and E. 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.

[26] Y. Marcus, E. Heilman, and S. Goldberg. Low-resource eclipse attacks on ethereum's peer-to-peer network. IACR Cryptology ePrint Archive, 2018:236, 2018.

[27] M. Mirkin, Y. Ji, J. Pang, A. Klages-Mundt, I. Eyal, and A. Juels. Bdos: Blockchain denial of service, 2019.

[28] D. Pérez and B. 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.

[29] M. Saad, L. Njilla, C. A. Kamhoua, J. Kim, D. Nyang, and A. 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.

[30] M. Tran, I. Choi, G. J. Moon, A. V. Vu, and M. S. 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.

[31] Y. Wang, Y. Tang, K. Li, W. Ding, and Z. Yang. Understanding ethereum mempool security under asymmetric dos by symbolized stateful fuzzing. In D. Balzarotti and W. Xu, editors, 33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024. USENIX Association, 2024.

[32] A. Yaish, K. Qin, L. Zhou, A. Zohar, and A. Gervais. Speculative denial-of-service attacks in ethereum. In D. Balzarotti and W. Xu, editors, 33rd USENIX Security Symposium, Philadelphia, PA, USA, August 14-16, 2024. USENIX Association, 2024.

Appendix A. Proof of Theorem 6.7

A.1. Relevant Theorems and Proofs

Before proving, we first build a useful analytical model of Algorithm 1. Specifically, we model the insertion to the ordered list in Line 4 in the algorithm by a sorting network.

The sorting network consists of conditional swaps or `cswap` defined as follows.

Definition A.1. Given two (unordered) input transactions tx_0, tx_1 and a set of transactions st, a `cswap` function produces as output a list of the two input transactions ordered by their $score_{\text{AP}}$ w.r.t. st. Formally,

$$\begin{aligned} \texttt{cswap}_{st}(tx_0, tx_1) &= tx_3, tx_2 \\ \{tx_3, tx_2\} &= \{tx_0, tx_1\} \\ score_{\text{AP}}(tx_3, st) &\geq score_{\text{AP}}(tx_2, st) \end{aligned} \tag{17}$$

Given a set of transactions txs and a transaction set st, function $x(txs)$ returns the transaction of the maximal $score_{\text{AP}}$ w.r.t. st, and function $n(txs)$ returns the transaction of the minimal $score_{\text{AP}}$ w.r.t. st.

We also define pairwise order insensitivity.

Definition A.2 (Pairwise order insensitivity). A mempool is admission-order insensitive if.f. for any initial state, denoted by st_0, and any two transactions arriving in order, tx_a, tx_b (i.e., tx_b arriving after tx_a), the end state reached by st_0 admitting tx_a, tx_b is the same as the state reached by st_0 admitting tx_b, tx_a. Formally,

$$\begin{aligned} &\forall st_0, tx_a, tx_b, \\ f(\langle st_0, \varnothing\rangle, tx_a, tx_b) &= f(\langle st_0, \varnothing\rangle, tx_b, tx_a) \end{aligned} \tag{18}$$

Lemma A.3. A mempool running Algorithm 1 is pair-wise order insensitive as in Definition A.2.

Proof We prove the theorem by considering the setup specified by Definition 6.1 and proving Equations 19.

Suppose a mempool running Algorithm 1 is of state st_0, in which the transactions of the lowest $score_{\text{AP}}$ and the second lowest $score_{\text{AP}}$ are denoted by tx_0 and tx_1, respectively. That is,

$$\begin{aligned} score_{\text{AP}}(tx_0, st_0) &= \min_{tx \in st_0} score_{\text{AP}}(tx, st_0) \\ score_{\text{AP}}(tx_1, st_0) &= \min_{tx \in st_0 \setminus \{tx_0\}} score_{\text{AP}}(tx, st_0) \end{aligned} \tag{19}$$

Now consider the timeline A in which the mempool receives two transactions in the following order: tx_a arriving before tx_b. In this timeline, Algorithm 1 would insert the arriving transactions tx_a, tx_b to the internal ordered list (see Line 4. In the model of a sorting network, it essentially runs `cswap()` three times as follows.

A1) Initially, Algorithm 1 runs `cswap()` on tx_0, tx_a, because $score_{\text{AP}}(tx_0, st_0) < score_{\text{AP}}(tx_1, st_0)$. That is:

$$\begin{aligned} \texttt{cswap}(tx_0, tx_a) &= tx_3, tx_2 \\ score_{\text{AP}}(tx_3, st_0 \cup \{tx_a\}) &\geq score_{\text{AP}}(tx_2, st_0 \cup \{tx_a\}) \\ tx_3 &= x(tx_0, tx_a) \\ tx_2 &= n(tx_0, tx_a) \end{aligned} \tag{20}$$

Assuming st_0 is full, it is tx_2 that is evicted from the mempool.

A2) The algorithm runs the second `cswap`:

$$\begin{aligned} \texttt{cswap}(tx_1, tx_b) &= tx_5, tx_4 \\ st_1 &= st_0 \cup \{tx_a\} \setminus \{tx_2\} \\ score_{\text{AP}}(tx_5, st_1 \cup \{tx_b\}) &\geq score_{\text{AP}}(tx_4, st_1 \cup \{tx_b\}) \\ tx_5 &= x(tx_1, tx_b) \\ tx_4 &= n(tx_1, tx_b) \end{aligned} \tag{21}$$

A3) The algorithm runs the third `cswap`:

$$\begin{aligned} \texttt{cswap}(tx_4, tx_3) &= tx_7, tx_6 \\ score_{\text{AP}}(tx_7, st_1 \cup \{tx_b\}) &\geq score_{\text{AP}}(tx_6, st_1 \cup \{tx_b\}) \\ tx_7 &= x(tx_4, tx_3) \\ tx_6 &= n(tx_4, tx_3) \end{aligned} \tag{22}$$

Transaction tx_6 is evicted from the mempool, leading to the end state in Equation 23:

$$st_2 = st_0 \cup \{tx_a, tx_b\} \setminus \{tx_2, tx_6\} \tag{23}$$

$$\{tx_2, tx_6\} = \{n(tx_0, tx_a), n(tx_b, tx_1, x(tx_0, tx_a))\} \tag{24}$$

From Equations 20, 21, and 22, we can derive Equation 24 as above.

Now consider timeline B in which the mempool has the same initial state st_0 and the same two arriving transactions except that they arrive in opposite order: tx_b before tx_a. By similar analysis and by symmetry, we can derive the end state as follows:

$$st'_2 = st_0 \cup \{tx_b, tx_a\} \setminus \{tx'_2, tx'_6\} \tag{25}$$

$$\begin{aligned} tx'_2 &= n(tx_0, tx_b) \\ tx'_6 &= n(tx_a, tx_1, x(tx_0, tx_b)) \\ \{tx'_2, tx'_6\} &= \{n(tx_0, tx_b), n(tx_a, tx_1, x(tx_0, tx_b))\} \end{aligned} \tag{26}$$

In the following, we prove $\{tx_2, tx_6\} = \{tx'_2, tx'_6\}$ by considering the three cases:

Case C1): $score_{\text{AP}}(tx_a, st_0) \leq score_{\text{AP}}(tx_0, st_0)$. Thus, $n(tx_0, tx_a) = tx_a$ and $x(tx_0, tx_a) = tx_0$. Because $n(tx_0, tx_1) = tx_0$, we can have:

$$\begin{aligned} \{tx_2, tx_6\} &= \{tx_a, n(tx_b, tx_0)\} \\ \{tx'_2, tx'_6\} &= \{n(tx_b, tx_0), tx_a\} = \{tx_2, tx_6\} \end{aligned}$$

Case C2): $score_{\text{AP}}(tx_b, st_0) \leq score_{\text{AP}}(tx_0, st_0)$. Thus, $n(tx_0, tx_b) = tx_b$ and $x(tx_0, tx_b) = tx_0$. Because $n(tx_0, tx_1) = tx_0$, we can have:

$$\begin{aligned} \{tx_2, tx_6\} &= \{n(tx_a, tx_0), tx_b\} \\ \{tx'_2, tx'_6\} &= \{tx_b, n(tx_a, tx_0)\} = \{tx_2, tx_6\} \end{aligned}$$

TABLE 6: Proof of order insensitivity: Equality between end state $dc = \{tx_2, tx_6\}$ under timeline tx_a, tx_b and $dc' = \{tx'_2, tx'_6\}$ under timeline tx_b, tx_a. In gray means outside the mempool.

	$score_{\text{AP}}(tx_a, st_0) \leq score_{\text{AP}}(tx_0, st_0)$	$score_{\text{AP}}(tx_b, st_0) \leq score_{\text{AP}}(tx_0, st_0)$	$score_{\text{AP}}(tx_a, st_0) > score_{\text{AP}}(tx_0, st_0) \wedge score_{\text{AP}}(tx_b, st_0) > score_{\text{AP}}(tx_0, st_0)$
$tx_2 = n(tx_0, tx_a)$ $tx_6 = n(tx_b, tx_1, x(tx_0, tx_a))$	tx_a $n(tx_b, tx_0)$	$n(tx_a, tx_0)$ tx_b	0 $n(tx_b, tx_1, tx_a)$
$tx'_2 = n(tx_0, tx_b)$ $tx'_6 = n(tx_a, tx_1, x(tx_0, tx_b))$	$n(tx_b, tx_0)$ tx_a	tx_b $n(tx_a, tx_0)$	0 $n(tx_b, tx_1, tx_a)$

Case C3): $score_{\text{AP}}(tx_a, st_0) > score_{\text{AP}}(tx_0, st_0) \wedge score_{\text{AP}}(tx_b, st_0) > score_{\text{AP}}(tx_0, st_0)$. Thus, $n(tx_0, tx_a) = tx_0$, $n(tx_0, tx_b) = tx_0$, $x(tx_0, tx_a) = tx_a$, and $x(tx_0, tx_b) = tx_b$.

$$\begin{aligned} \{tx_2, tx_6\} &= \{tx_0, n(tx_b, tx_1, tx_a)\} \\ \{tx'_2, tx'_6\} &= \{tx_0, n(tx_b, tx_1, tx_a)\} = \{tx_2, tx_6\} \end{aligned}$$

The three cases are summarized in Table 6. Proven.

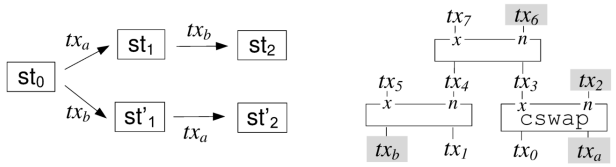


Figure 11: Order-insensitive mempool and a sorting-network model of transaction admission.

A.2. Main Proof of Theorem 6.7

We now present the proof of Theorem 6.7.

Proof Consider a timeline $f(\langle st_0, \varnothing\rangle), \vec{as} \Rightarrow \langle st_n, dc_n\rangle$ and another timeline from $\vec{as}$, that is, $f(\langle st_0, \varnothing\rangle), \vec{as''} \Rightarrow \langle st''_n, dc''_n\rangle$. $\vec{as''}$ is reordered from $\vec{as}$, while preserving the happen-before partial order.

Consider the original timeline $\vec{as}$ (full description in Equation 10) and a destination timeline $\vec{as''}$ (in Equation 11). We can construct a procedure (i.e., a sorting network) as follows that converts $\vec{as}$ to $\vec{as''}$.

Denote $\vec{as}$ and $\vec{as''}$ by $\vec{as'_0}$ and $\vec{as'_n}$. For $\forall k \in [0, n-1]$, consider converting $\vec{as'_k}$ to $\vec{as'_{k+1}}$ as follows:

- Step 1) Find the first transaction of index i on $\vec{as'_k}$ that differs from the i-th transaction on $\vec{as''}$. That is, $\vec{as'_k}.at(i) \neq \vec{as''}.at(i)$.
- Step 2) If $\vec{as'_k}.at(i)$ is not the parent of $\vec{as'_k}.at(i+1)$, swap $\vec{as'_k}.at(i)$ and $\vec{as'_k}.at(i+1)$. Otherwise, it moves the reference to the next position ($j = i+1$) and tries swapping $\vec{as'_k}.at(j)$ and $\vec{as'_k}.at(j+1)$. It keeps incrementing j until it successfully conducts a swap.
- Step 3) By incrementing j, it keeps doing swaps until transaction $\vec{as'_k}.at(i)$ is ordered after transaction $\vec{as''}.at(i)$.
- Step 4) Repeat Steps 1), 2) and 3) with the same i, until $\vec{as'_k}.at(i)$ equals $\vec{as''}.at(i)$. In the end of Step 3), we denote the transaction list by $\vec{as'_{k+1}}$.
- Step 5) Repeat Steps 1), 2), 3), and 4) after enough iterations, say $n-1$ iterations, until $\vec{as'_n} = \vec{as''}$.

Consider each swap step in Step 2) in the above procedure. That is, given the current transaction list $\vec{as'_{k^j}}$, it swaps the j-th and $j+1$-th transactions on this list, resulting in $\vec{as'_{k^{j+1}}}$. Because the procedure above swap transactions without violating the parent-before relationship, and due to Lemma A.3, given the same initial state $\langle st_0, dc_0\rangle$, replaying transactions in $\vec{as'_{k^j}}$ leads to the same end state with replaying transactions in $\vec{as'_{k^{j+1}}}$.

By induction, the overall swap procedure from the beginning list $\vec{as}$ to the end list $\vec{as''}$ does not change the mempool end state, that is, $\langle st_n, dc_n\rangle = \langle st''_n, dc''_n\rangle$.

Appendix B. Proof of Theorem 6.6

First, we prove Lemma 6.5.

Proof We prove this lemma by contradiction. Assume $tx \in st_i$ is turned into a future transaction in st_{i+1}.

There must be an ancestor of tx, say tx', that is evicted from st_{i+1} by Algorithm 1. Due to Line 5, tx' must be ordered the last on $\vec{st}$. That is, tx' must be ordered after tx on $\vec{st}$. However, $tx.nonce > tx'.nonce$ and, according to the break-even rules in TORDER() (described in § 5.2), tx' is ordered before tx on the total order. Contradiction.

Second, we present and prove a relevant Lemma B.1 before proving Theorem 6.6.

Lemma B.1 (Independence of $score_{\text{AP}}$). Suppose a mempool running Algorithm 1 with $score_{\text{AP}}$ admits an arriving transaction ta_i and transitions its state from st_i to st_{i+1}. That is, $\text{ADTX}(st_i, ta_i) \rightarrow st_{i+1}, te_i$ as in Equation 1. For the set of transactions residing in the mempool before and after the admission (i.e., $st_i \cap st_{i+1}$), their $score_{\text{AP}}$ do not change. Formally,

$$\begin{aligned} &\forall tx \in st_{i+1} \cap st_i = st_{i+1} \setminus \{ta_i\} \\ &score_{\text{AP}}(tx, st_i) \equiv score_{\text{AP}}(tx, st_{i+1}) \quad (27) \end{aligned}$$

Proof We prove the lemma by two properties: 1) Given a transaction tx and a set st, $score_{\text{AP}}(tx, st)$ depends only on tx's ancestors in st. 2) Given state transition from st_i to st_{i+1}, if $tx \in st_i \cap st_{i+1}$, any of tx's ancestors, say tx', must also belong to $st_i \cap st_{i+1}$.

Property 1) can be easily derived from Equation 8.

We prove Property 2) by contradiction. Given a $tx \in st_i \cap st_{i+1}$, assume tx' is tx's ancestor, and $tx' \notin st_i \cap st_{i+1}$

. Consider three sub-cases: 2a) $tx' \notin st_i \wedge tx' \notin st_{i+1}$. 2b) $tx' \notin st_i \wedge tx' \in st_{i+1}$. 2c) $tx' \in st_i \wedge tx' \notin st_{i+1}$.

In sub-case 2a), tx must be a future transaction in both st_i and st_{I+1}, which contradicts the setting that our mempool stores only pending transactions.

In sub-case 2b), tx must be a future transaction in st_i. Contradiction.

In sub-case 2c), Algorithm 1 must evict $te_i = tx'$, and tx' is an ancestor to tx. In other words, Algorithm 1 under $score_{\text{AP}}$ must turn tx into a future transaction in st_{i+1}, which contradicts with Lemma 6.5.

Overall, Property 2) holds.

Third, we now present the proof of Theorem 6.6.

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 prove the following equation:

$$\sum_{tx \in st_{i+1}} score_{\text{AP}}(tx, st_{i+1}) \geq \sum_{tx \in st_i} score_{\text{AP}}(tx, st_i) \quad (28)$$

To start with, denote by st the set of transactions that exist in both st_i and st_{i+1}. That is, $st = st_i \cap st_{i+1}$. Due to the score independence (i.e., Lemma B.1), we have

$$\sum_{tx \in st} score_{\text{AP}}(tx, st_i) = \sum_{tx \in st} score_{\text{AP}}(tx, st_{i+1})$$

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

$$\begin{aligned}\sum_{tx \in st_i} score_{\text{AP}}(tx, st_i) &= \sum_{tx \in st} score_{\text{AP}}(tx, st_i) \\ &= \sum_{tx \in st} score_{\text{AP}}(tx, st_{i+1}) \\ &= \sum_{tx \in st_{i+1}} score_{\text{AP}}(tx, st_{i+1})\end{aligned}$$

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

$$\begin{aligned}&\sum_{tx \in st_{i+1}} score_{\text{AP}}(tx, st_{i+1}) \\ =& \sum_{tx \in st_i \cup \{ta_i\}} score_{\text{AP}}(tx, st_{i+1}) \\ =& \sum_{tx \in st_i} score_{\text{AP}}(tx, st_{i+1}) + score_{\text{AP}}(ta_i, st_{i+1}) \\ =& \sum_{tx \in st} score_{\text{AP}}(tx, st_{i+1}) + score_{\text{AP}}(ta_i, st_{i+1}) \\ =& \sum_{tx \in st_i} score_{\text{AP}}(tx, st_i) + score_{\text{AP}}(ta_i, st_{i+1}) \\ \geq& \sum_{tx \in st_i} score_{\text{AP}}(tx, st_i)\end{aligned}$$

In Case 3), $st_{i+1} \setminus \{ta_i\} = st_i \setminus \{te_i\} = st$. Applying Line 7 in Algorithm 1, we can derive the following:

$$\begin{aligned}&\sum_{tx \in st_{i+1}} score_{\text{AP}}(tx, st_{i+1}) \\ =& \sum_{tx \in st \cup \{ta_i\}} score_{\text{AP}}(tx, st_{i+1}) \\ =& \sum_{tx \in st} score_{\text{AP}}(tx, st_{i+1}) + score_{\text{AP}}(ta_i, st_{i+1}) \\ =& \sum_{tx \in st} score_{\text{AP}}(tx, st_i) + score_{\text{AP}}(ta_i, st_{i+1}) \\ \geq& \sum_{tx \in st} score_{\text{AP}}(tx, st_i) + score_{\text{AP}}(te_i, st_i) \\ =& \sum_{tx \in st \cup \{te_i\}} score_{\text{AP}}(tx, st_i) \\ =& \sum_{tx \in st_i} score_{\text{AP}}(tx, st_i)\end{aligned}$$

Therefore, in all three cases, Equation 28 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 28 for $i \in [0, n-1]$ and prove the sum of $score_{\text{AP}}$ monotonically increases.

Appendix C. Proof of Lemma 6.9

Proof Given timeline $f(\langle st_0, \varnothing \rangle, \vec{as}) \Rightarrow \langle st_n, dc_n \rangle$, consider any $k \in [0, n]$ and the associated admission step: $\text{ADTX}(tx_k, st_k) = te_k, st_{k+1}$. Because of Line 7 in Algorithm 1 and Equation 27, if tx_k is declined, we have $st_k = st_{k+1}$. Thus,

$$\min_{tx \in st_k} score_{\text{AP}}(tx, st_k) = \min_{tx \in st_{k+1}} score_{\text{AP}}(tx, st_{k+1}) \quad (29)$$

If tx_k is admitted, $te_k = txm(st_k)$. The following holds:

$$\begin{aligned}&\min_{tx \in st_k} score_{\text{AP}}(tx, st_k) \\ =& \min(\min_{\substack{tx \in st_k \setminus \\ \{txm(st_k)\}}} (score_{\text{AP}}(tx, st_k)), score_{\text{AP}}(txm(st_k), st_k)) \\ \leq& \min(\min_{\substack{tx \in st_k \setminus \\ \{txm(st_k)\}}} (score_{\text{AP}}(tx, st_k)), score_{\text{AP}}(tx_k, st_k)) \\ =& \min(\min_{\substack{tx \in st_{k+1} \setminus \\ \{tx_k\}}} (score_{\text{AP}}(tx, st_{k+1})), score_{\text{AP}}(tx_k, st_{k+1})) \\ =& \min_{tx \in st_{k+1}} score_{\text{AP}}(tx, st_{k+1})\end{aligned}$$

Induction from $k = i$ to $k = j - 1$ leads to Equation 15.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

The paper presents a new security definition for evaluating defense against asymmetric DoS attacks in Mempools. It also introduces formal definitions for security and presents a framework called SAFERAD for securing Mempools against eviction and locking attacks. It also presents the practicality of the approach by evaluating their approach on Ethereum client Geth. The overall takeaway is that the paper presents formal evidence that SAFERAD offers security with minimal overhead.

D.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

D.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. It provides a formal approach to study and evaluate security defenses against asymmetric DoS attacks in Mempools. Formal justification adds value to the paper and provides justification for the soundness and correctness of the overall approach.
2) The paper creates a new tool to enable future science. The paper presents the tool SAFERAD which offers security with minimal overhead. The tool will be available to serve as a platform for future research.

D.4. Noteworthy Concerns

1) The paper is hard to follow and makes it difficult to assess the correctness and soundness of the overall approach. A formal description of the threat model and all the formal definitions presented in Section 4 should lead to making the reader understand about the formal justifications presented in the paper and should connect the formal justifications with the overall results.
2) The paper lacks the comparison with existing work and discussion about how previous attacks and solutions fit into the proposed framework in Section 1. For example, discussion related to attacks discovered in MPFUZZ [31] and how attacks discovered in DETER [25] and MemPurge [32] fits with the proposed model.
3) The paper lacks proper discussion about the limitations and to what level the proposed framework can be generalized. For example, the security analysis presented in the paper only considers intra-block attacks and what extensions it would need to tackle the attacks where interleaving of blocks and transactions are allowed. Also, the defense seems to be applicable only to the valid transaction pool and the security analysis is more inclined towards Ethereum, which makes it hard to analyze the generalization of the overall approach.
4) The paper also lacks proper evaluation due to the fact of making the experiments abstract instead of actually executing the transactions and it does not capture the adaptive behavior of the transaction senders. This makes it hard to evaluate the theoretical results through the experiments.

Appendix E. Response to the Meta-Review

In Section § 4.1, we discuss the correctness of our definitions and describe how they capture known attacks [25], [31], [32] and all possible asymmetric mempool-DoS attacks. The pre-checks in Algorithm 1 (i.e., Line 1) are compatible with and has integrated the rule-based defenses to specifically mitigate known attacks, such as DETER and mpfuzz [11], [25]. § 9.2 presents the evaluation of SAFERAD's performance when users resubmit transactions upon observing transaction exclusion from blocks. The paper also discusses the design rational of replaying transactions without actual execution, generalizability to non-Ethereum blockchains, and the limitation of the current security analysis.