Speculative Denial-of-Service Attacks in Ethereum

Aviv Yaish
The Hebrew University

Kaihua Qin
Imperial College London, UC Berkeley RDI

Liyi Zhou
Imperial College London, UC Berkeley RDI

Aviv Zohar
The Hebrew University

Arthur Gervais
University College London, UC Berkeley RDI

Abstract

Transaction fees compensate actors for resources expended on transactions and can only be charged from transactions included in blocks. But, the expressiveness of Turing-complete contracts implies that verifying if transactions can be included requires executing them on the current blockchain state.

In this work, we show that adversaries can craft malicious transactions that decouple the work imposed on blockchain actors from the compensation offered in return. We introduce three attacks: *(i)* ConditionalExhaust, a conditional resource exhaustion attack (REA) against blockchain actors. *(ii)* MemPurge, an attack for evicting transactions from actors' mempools. *(iii)* GhostTX, an attack on the reputation system used in Ethereum's proposer-builder separation (PBS) ecosystem.

We evaluate our attacks on an Ethereum testnet and find that by combining ConditionalExhaust and MemPurge, adversaries can simultaneously burden victims' computational resources and clog their mempools to the point where victims are unable to include transactions in blocks. Thus, victims create empty blocks, thereby hurting the system's liveness. The attack's expected cost is $376, but becomes cheaper if adversaries are validators. For other attackers, costs decrease if censorship is prevalent in the network.

ConditionalExhaust and MemPurge are made possible by inherent features of Turing-complete blockchains, and potential mitigations may result in reducing a ledger's scalability.

1 Introduction

Blockchains such as Ethereum rely on highly expressive smart contract languages to enable the creation of a rich and diverse decentralized finance (DeFi) ecosystem. The flexibility and the open nature of these systems pose a risk: users may deploy contracts that consume large amounts of computational resources, and may overwhelm all nodes that validate the blockchain with expensive computations. The answer Ethereum's designers have put forth is to run all computations with a restricted budget of operations. Each computational action costs a certain amount of "gas", and a strict gas limit is placed on all transactions. Furthermore, users are required to pay fees per unit of gas that they consume, making it expensive for attackers to overload blockchain nodes.

This work. We show that the gas mechanism is insufficient to protect nodes from denial-of-service (DoS) attacks. By expanding on the insights of notable previous works [36, 44, 47], we present several effective attacks against Go Ethereum (geth)-based clients, the most prevalent Ethereum client. While the attacks of previous works are mitigated, our attacks circumvent existing defenses, and result in severely degraded performance of victim nodes. We evaluate our attacks on a local testnet and show that by sending 140 transactions, attackers can prevent victims from mining *any* transaction.

We leverage several key insights to construct our attacks, each insight separately allows us to waste victims' resources at a minimal cost: *1.* Ethereum's partitioning of the block creation process to several roles (*searchers*, *builders*, *relays*, and *proposers*) forces some nodes to execute transactions heuristically or speculatively. *2.* The behavior of smart contract code can be made highly dependent on context, i.e., on the state of other smart contracts and accounts. *3.* Some nodes selectively adopt external censorship policies on transactions. These insights allow creating transactions that are resource intensive when executed speculatively, but are excluded from the blockchain. Furthermore, even if transactions are not executed, they occupy limited memory pool (mempool) space, that could be used for more profitable ones.

Motivation. Our attacks can be launched at a low cost by adversarial actors such as builders and staking pools to improve their revenue while hurting their competitors'. In particular, they allow an attacker to reserve profitable transactions to itself by preventing competitors from including them in their blocks. Our attacks can also confer an advantage to adversaries with respect to common time-sensitive blockchain mechanisms, such as voting protocols [13, 55], payment channels that rely on deadlines [49], and lending platforms [56].

Our attacks We show three attacks: *1.* The *ConditionalExhaust* attack, summarized in Fig. 2, involves creating transactions that execute computationally intensive code conditional on the executing validator's identity, thereby making sure that these expensive computations are only performed if the validator *cannot* include the transactions in a block. This can happen if, for example, transactions culminate with an interaction with a sanctioned address, which the validator censors to be compliant with the law. *2.* The *MemPurge* attack, depicted in Fig. 3, is distinct from ConditionalExhaust and applies to cases where transactions are not executed. In particular, nodes heuristically verify incoming transactions before adding them to their mempools, without executing them. The attack cheaply evicts honest transactions from victims' mempools by creating chains of transactions that seem valid at first, but become invalid after executing each chain's initial transaction. *3.* In the *GhostTX* attack, presented in Fig. 6, an attacker crafts transactions that appear lucrative to searchers and builders, yet that cannot be included in blocks, thereby harming their standing in Flashbots' reputation mechanism.

Our attacks demonstrate that the sensitivity of transaction validity to execution context exposes actors to adversarial manipulations. This is in spite of geth and the ecosystem at large accumulating a layer of protections that were developed to curtail the high incidence of DoS attacks in Ethereum [8, 43, 44, 47, 50]. In particular, our attacks circumvent the following protective heuristics: *1.* Transactions are verified with stringent out-of-consensus heuristics to ensure senders can cover all associated fees, even when accounting for previously received pending transactions by the same senders. *2.* The per-address number of transactions is limited. *3.* A single transaction may be verified multiple times by actors involved in each step of the block-creation process (searchers, builders, relays, and validators), and passed to the next one only if valid. *4.* Victims can broadcast transactions to the network to ensure that an attack is not free.

Mitigations for these attacks may require limiting blockchain scalability, quality of service, and the revenue of actors such as builders and proposers.

Our contributions. In summary, our contributions are:

- **ConditionalExhaust.** We introduce a novel REA vector, which becomes more cost-effective when targeting victims that actively engage in transaction censorship, such as block builders and validators. By developing a best-effort tool to craft resource-exhausting transactions, we demonstrate that an attacker can prevent a victim from including transactions in blocks by sending only 140 attack transactions which exhaust the victim's computational resources.
- **MemPurge.** We propose the MemPurge attack, which can efficiently evict transactions from victims' mempools. We assess its performance and show it bypasses mitigations put in place to prevent related previous attacks.
- **GhostTX.** This attack compels block builders to include transactions that result in resource waste for actors and reputational damage for searchers who supply builders with tainted bundles. To the best of our knowledge, this is the first attack targeting the PBS ecosystem.
- **Empirical evaluation.** We evaluate our attacks by employing a testing framework that sets up a local testnet and analyzing relevant data, including average resource consumption of transactions, the typical mempool state, and searcher reputation. We find that the costs of the attacks diminish if the adversary is a validator, or if a greater proportion of actors engage in censorship.

Disclosure. Our work was disclosed to the Ethereum Foundation (EF) and the Flashbots company. The authors provided both the EF and the Flashbots company with a draft of this paper, together with implementations of all attacks, code that executes them on a private local testnet, and suggestions for mitigations. Both acknowledged the respective issues quickly, and awarded the authors with bounties.

2 Background

Censorship. Cryptocurrency mixers allow users to obfuscate their tokens' original ownership. The potential use of mixers for illicit purposes such as money laundering caught the attention of law enforcement agencies: on August '22, the United States (US) Office of Foreign Assets Control (OFAC) sanctioned the Tornado Cash (TC) mixer [52]. This action restricts interaction with TC, and includes the addresses of TC's Ethereum contracts on OFAC's Specially Designated Nationals and Blocked Persons (SDN) list. Consequently, actors looking to abide by US law started censoring TC-related transactions within blocks [52]. The consequences of OFAC's sanctions have rapidly emerged, with prominent Ethereum actors being OFAC-compliant [42], raising concerns within the Ethereum ecosystem [37, 40, 54].

Proposer-builder separation (PBS). Various blockchain actors, summarized in Fig. 1, work together to extract profits known as miner-extractable value (MEV). MEV may arise from arbitrage opportunities due to price disparities between DeFi platforms, and can also be maliciously extracted by leveraging public and private information, e.g., by front running transactions heard on the peer to peer (p2p) layer [12]. In this landscape, *searchers* specialize in identifying MEV opportunities and assembling transaction bundles exploiting them. Bundles are sent to *builders*, who use them together with p2p transactions to construct profitable blocks. *Relays* verify and share the most lucrative blocks with the validator designated as the upcoming block *proposer* using the MEV-Boost program [18]. Proposers may use relayed blocks or construct blocks themselves from transactions sent on the p2p

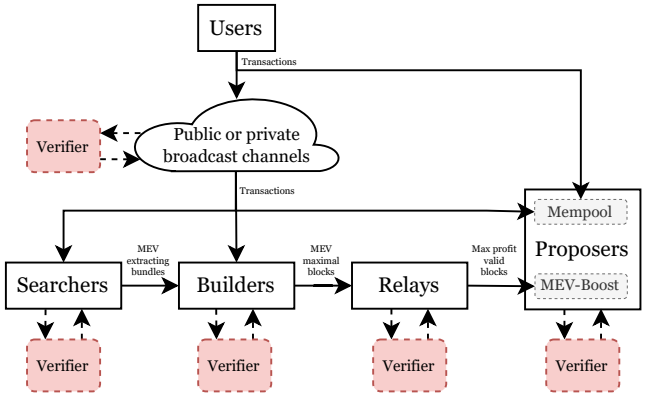

Figure 1: Overview of Ethereum's PBS ecosystem actors.

layer or directly to them, these are stored in a data structure called the *mempool*. The division of labor between builders and proposers is known as *PBS*. PBS has been advanced as a panacea for Ethereum's censorship woes [15, 18]. Yet, empirical evidence shows that Ethereum builders and relays engage in censorship [42]. In fact, Flashbots' builder client facilitates compliance with custom blacklists [21], and its "example" list was based on OFAC's SDN list until March '23 [33].

3 Model

Our model follows Ethereum, and captures most popular cryptocurrencies that support Turing-complete smart contracts.

Blockchain. Transactions are processed in batches called *blocks*. An underlying consensus mechanism elects a leader for each block in an i.i.d. manner, who then chooses the transactions to include in its block. Leaders are assumed to select transactions greedily, by their fees [26]. In proof-of-work (PoW) mechanisms such as Bitcoin's, leaders are elected among so-called *miners*. Under proof-of-stake (PoS) mechanisms like Ethereum's, *validators* are chosen with a probability equal to their share of stake in the system [4]. For conciseness, we use the term validator for both. The blockchain supports the distributed execution of programs called smart contracts, written in a Turing-complete virtual machine (VM) language. The complexity of basic VM instructions, also called *opcodes*, is fixed and measured in a unit called *gas*. Moreover, blocks have an upper *gas limit*.

Transactions. Users can interact with the cryptocurrency by creating *transactions* that specify, in code, actions they wish to execute, primarily: *1.* Transfer funds between two addresses. *2.* Create (e.g., *deploy*) a smart contract. *3.* Invoke a function of a deployed contract. A transaction τ is identified by: *1.* Its *nonce* τ_n, which is a serial number that determines the inclusion order of all transactions sent by the same user, *2.* The *value* τ_v it transfers to the recipient's address, *3.* Its *fee* or *gas price* per unit of gas τ_f, which can be collected by the first validator to include the transaction in a block.

Transaction execution. Transactions are executed opcode by opcode, until either there are no opcodes left, or senders' balances cannot cover the gas required to continue execution.

Pending and future transactions. A transaction τ by user u is considered *pending* for inclusion in the next block if its nonce is larger by 1 than the nonce of u's last accepted transaction τ', whether τ' is included in the same or previous blocks [38]. If transactions are not pending or accepted, they are called *future* transactions. Nodes store pending and future transactions in a data structure called a *mempool*, or *txpool* in Ethereum's nomenclature. For generality, we use the former.

Fee bumping. Mempool transactions can be replaced by transactions with an equal nonce and a fee larger by a minimal node-chosen amount $x \geq 1$, an act called *fee bumping*.

Transaction gossip protocol. The blockchain's p2p protocol has a message for requesting a list of transactions identified by their hashes from peers. The protocol also has a message for propagating newly heard-of transactions to peers. The corresponding Ethereum messages are *GetPooledTransactions* and *NewPooledTransactionHashes* [14, 41].

Actors

Blockchain users. Users can create multiple addresses, and use them to sign transactions that are then broadcast to nodes participating in the network over the p2p layer.

Sanctioned entity. There is at least one sanctioned entity active on the system, meaning that some of the cryptocurrency's validators actively censor the entity and abstain from including transactions that interact with it in their blocks. Let σ be the sanctioned entity's address, S be the set of validators censoring σ, and $\alpha \in [0, 1]$ be the set's total fraction of stake. Both S and α are assumed to be estimated by an attacker using public blockchain data. In Ethereum, each validator's stake is public knowledge and fixed for a certain period of time, thus the set of censoring validators can be accurately estimated, provided validators do not alter their censorship policies.

Censorship method. The compliance of a transaction with a node's censorship policy is verified by: *1.* checking hard-coded transaction fields to be free from sanctioned entities (e.g., the transaction's recipient address), *2.* if all are valid, the transaction is executed on the latest blockchain state and its execution is verified to be free from forbidden interactions.

Furthermore, all nodes broadcast incoming valid transactions to their peers, whether they are compliant or not. As censoring nodes broadcast non-compliant transactions, would-be attackers are weakened: their transactions will reach non-censoring nodes, and therefore may enter blocks and incur fees.

Remark 1. *We note that this censorship method is adopted by ecosystem actors [21], and any other method may expose actors to attacks. Due to the halting problem, it is impossible to have foreknowledge of a general transaction's execution path, implying that execution is the only method that guarantees transaction compliance. If a censoring actor does not execute transactions to ensure compliance, it can be attacked by sending non-compliant transactions that the actor will include in a block or bundle, thereby exposing itself to litigation. Moreover, online sources that track censorship in Ethereum show that OFAC compliance is common among censoring actors, and publish the addresses of compliant actors [16, 33, 42]. Furthermore, Ethereum validator addresses are fixed until withdrawal. This means that for adversaries wishing to target a broad range of victims, a good choice for S is the set of OFAC-compliant actors, and for σ is one of TC's addresses (or other OFAC-sanctioned addresses).*

Adversary. To exhibit the strength of our attacks, we consider a weak adversary $\mathcal{A}$ who interacts with the system by creating and sending transactions sent using the transaction gossip protocol, and does not partake in the underlying consensus. Moreover, the adversary derives its strategies by relying on its partial view of the Ethereum network, considering only its single node to estimate network properties, such as the fees paid by accepted transactions. In terms of processing capabilities, we assume the attacker can send transactions at a similar rate as an average validator. The attacker cannot interfere with its victims' network communications. Although outside the model, throughout the work we also outline how block proposers can execute our attacks at nearly no cost.

4 The ConditionalExhaust Attack

We now present a REA we call *ConditionalExhaust*, which allows an adversary to cause actors that execute transactions (such as block builders and proposers) to create empty blocks and to needlessly expend their resources. This is done by wasting their time on executing resource-consuming transactions that cannot be included in blocks and thus do not pay fees, rather than profitable "honest" transactions. We proceed with an overview of attack variants, followed by implementation details, and an evaluation of attack costs and impact.

ConditionalExhaust for adversarial proposers. If our adversary $\mathcal{A}$ is a block proposer, then it can attack actors such as searchers, builders, and relays. Although this is outside our model, we quickly describe the attack as a stepping stone toward a more interesting variant that can both 1. be executed by adversaries that are not proposers, and 2. target proposers. Intuitively, actors besides the upcoming proposer cannot know for certain which transactions will be included in the block, and in what order. If the adversary is scheduled to propose the upcoming block (the schedule of block proposers is publicly known in advance in Ethereum), it can spam the network with valid computationally intensive transactions which are generated from some pre-funded address. To prevent attack transactions from incurring high fees, the adversary should set the first transaction of its block to transfer all funds from the pre-funded address, to another address in its possession. Thus, while victims may execute the attacker's spam transactions and incur costs for doing so, all are invalidated by the upcoming block.

ConditionalExhaust for non-proposer attackers. If the attacker is not a block proposer, then it can attack sanction compliant builders and proposers, and can harm the blockchain's liveness if the latter are targeted. In the previous variant, our adversary used its ability to propose the upcoming block to cleverly include a transaction that invalidates the work of other actors. For the current variant, we assume that the adversary is not a proposer, meaning that on the one hand it can now target proposers, but that a new technique is required to invalidate our victims' work. Intuitively, compliant actors cannot create blocks that include transactions which interact with sanctioned entities, while a transaction's compliance cannot be verified without executing it. This allows an attacker to "trick" victims that censor a given entity to execute transactions that they cannot include in a block and thus cannot collect a fee from. These transactions interact with the sanctioned entity, but that are crafted to both:

1. Preclude trivially verifying whether they should be censored, thereby wasting the victims' resources.
2. Ensure that even if they are included in a block, the cost for the attacker will be minimal.

4.1 Attack Description

We now go over the second variant, with a graphical depiction given in Fig. 2. The attack advances in two phases.

Deployment phase. First, $\mathcal{A}$ deploys a smart contract with a single function that has two different control flows, incurring deployment costs of φ fees. When the function is invoked by a transaction, the flow is chosen according to the identity of the validator executing the transaction:

1. If the validator belongs to the set of censoring validators S, a conditional statement will trigger the execution of a computationally intensive branch of code which results in an interaction with the censored entity σ.

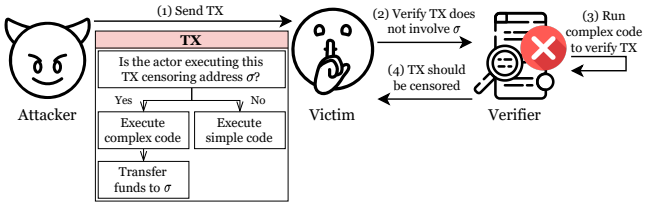

Figure 2: ConditionalExhaust is a conditional REA, in which an attacker creates transactions that invoke computationally complex code if the victim cannot include them in a block, for example due to its censoring policy.

2. Otherwise, a computationally simple branch will be executed, incurring fees equal to ϕ.

Execution phase. After deployment, the attack proceeds to the second phase. In it, the attacker creates multiple transactions that trigger the contract's single function. We note that if censoring actors discard non-compliant transactions from their mempools, an attack becomes substantially cheaper, as an attacker can re-send the same transaction again and again. If this transaction finds its way to a non-compliant party, it may be included in a block and cost the attacker the fees which are associated with the computationally simple branch. Due to nonce considerations, only one such transaction can be included in each block. If an attacker wishes to target actors who do not discard such transactions, the nonce of each consecutive attack transaction should increase by 1.

Correctness. Any actor in S that receives one of $\mathcal{A}$'s transactions will execute the intensive branch of the contract. Only when reaching the end of the code, the actor can observe that the transaction interacts with σ, and thus should be censored. As long as S indeed corresponds to actors that censor σ, then the computationally intensive branched will only be executed by those who cannot include the transaction in a block.

Implementation

A construction of an Ethereum contract that executes the attack is given in Listing 1. The novelty of the implementation lies in carefully designing transactions that have two flows, one intensive and the other not, where at the worst case only the fees for the simple flow are paid. Instead of minimizing the cost of the intensive flow, we only wish to maximize its resource consumption. To do so, we rely on inefficient constructs used in Ethereum.

Ethereum's state. Ethereum's implementation guidelines propose saving parts of the blockchain's state using the Merkle-Patricia trie data structure [38]. Although the exact

```
pragma solidity >=0.7.0 <0.9.0;
contract ConditionalExhaustCoinbaseVariant {
 mapping (address => bool) private _shouldDoS;
 /// @notice Creates a set of the validators to DoS.
 constructor() {
  _shouldDoS[AddressToDoS1] = true;
  // _shouldDoS[AddressToDoS2] = true;
  // ...
 }
 function DoS(uint32 i) external payable {
  bool shouldDoS = _shouldDoS[block.coinbase];
  assembly {
   if shouldDoS {
    // The computationally complex part of the TX:
    for { } gt(i, 0) { i := sub(i, 1) } {
     pop(extcodehash(xor(blockhash(number()), gas())))
    }
    // Replace "CensoredAddress" with your favorite
    // sanctioned address!
    pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))
   }
   stop()
  }
 }
}
```

Listing 1: An implementation of an Ethereum smart contract that facilitates the ConditionalExhaust attack, for an adversary who knows the addresses of censoring validators.

details are out of the work's scope, this structure is considered inefficient due to the amount of storage operations required for simple tasks, such as reading address balances [48,50]. Therefore, it is not surprising that DoS attacks relying on storage-heavy transactions have plagued Ethereum [8, 11, 50, 53, 58].

Inefficient opcodes. To devote most of the code's complexity to inefficient opcodes, we wrote most logic in Yul, a commonly used in-line assembly language [10, 45]. The contract's complexity is obtained by accessing random locations in Ethereum's state using inefficient storage operations. Specifically, we use the *EXTCODEHASH* opcode [39], which reads the code of a deployed contract and returns its hash.

Deriving randomness. Deriving a "good" source of randomness in a blockchain setting is challenging [6], and out of the scope of this work. For our purposes, a good approximation can be achieved by performing an exclusive or (XOR) operation between the current block's hash and the amount of gas remaining for the execution of the transaction. The former provides some basic pseudo-randomness that varies across blocks, while the latter modifies this randomness over the course of a single transaction's execution.

Coinbase variant. We call the attack described so far the *coinbase* variant. To summarize, the attack relies on adversaries having prior knowledge of the addresses of censoring validators S and of an address they are known to censor σ, and by setting these parameters in the attack contract, victims in S trigger a computationally expensive execution branch that culminates with a non-compliant transfer to σ which cannot

be included in a block, thereby assuring that attackers do not incur high fees. For adversaries wishing to target victims who censor different entities, the branch can end with multiple transfers, one to each entity, thus having a broader effect.

Blockheight variant. In the full version of the paper [57], we provide an implementation of a *blockheight* variant of the attack, that executes the complex branch if the current block's height is equal to an attacker-specified parameter. Both variants are functionally equivalent, given that in Ethereum: *1.* There are services for querying the schedule of upcoming validators (such as Flashbots' endpoint which returns a list of addresses for the current and upcoming epochs [24]), *2.* Validator addresses are fixed until withdrawal, *3.* The identity of censoring validators and the addresses which they censor are known [33, 42].

4.2 Evaluation

To empirically evaluate our attacks, we develop a framework that allows testing attacks on a testnet and measuring a given transaction's execution time in isolation. Our framework uses Flashbots' builder client [20], a geth fork that implements the censorship functionality described in Section 3. Our evaluation was done on a machine that exceeds Flashbots' official requirements [19]. These currently ask for a computer with a 4 core CPU operating at 2.8GHz, 16GB of RAM, and an SSD. Our testbed uses Ubuntu 20.04.2 LTS, an AMD Ryzen Threadripper 3990X CPU with 64 cores and 128 threads operating at 2.9GHz, 256GB of RAM, and NVMe SSDs.

4.2.1 Runtime Evaluation

Gas. A transaction deploying the coinbase variant consumes 120,750 gas units. If censoring validators execute an attack transaction, a code path which consumes a block's entire gas quota is executed, currently set at $3 \cdot 10^7$ units. When non-censoring validators execute the transaction, 23,628 gas units are consumed, only 12.5% more than the 21,000 units consumed by the most gas-efficient Ethereum transaction. We note that the larger the number of validators that should be attacked, more gas is required to deploy the contract. For example, if six validators are targeted instead of just one, 257,761 units are needed. On the other hand, the gas required to execute the simple code branch remains unchanged. In contrast, the contract for the blockheight-based variant of the attack does not rely on hard-coded victim addresses. Thus, deploying it has a fixed gas consumption of 97,885 units. As the contract is simpler, the gas consumption for transactions that are included in blocks is lower and equals 21,429 units.

Transaction creation & verification times. Our framework allows measuring the time needed to verify a given transaction in isolation. As a more complex blockchain state can increase transaction verification speed, we control for this and provide lower bounds by using a basic state consisting of a single block with a single transaction. Given this initial state, we created 10,000 different attack transactions. On average, a transaction was created and signed in $5.5 \cdot 10^{-5}$ seconds. Verifying an attack transaction required an average time of 0.1 ± 0.011 seconds when performed by the censoring validation software. In comparison, simple value transfers are validated in 0.001 seconds, on average. Thus, verification is $1972\times$ more time-consuming than transaction creation. This means that an attacker can keep up with a single victim even if the latter is in possession of hardware that is 1972 times more performant than that of the former. As the same transactions can be sent to the entire network, this logic holds no matter how many high-performance victims are targeted.

Attacking a testnet. In Ethereum, a block is created every 12 seconds, meaning that 120 ConditionalExhaust transactions can be verified, on average, between blocks. Evaluating the attack on a local private testnet set up on our testbed affirms that an attacker sending 140 transactions can exhaust a victim's resources to the point that it is unable to verify even a single honest transaction in time for including it in the next block. Even when letting the victim create 100 consecutive blocks, a one-shot attack consisting of 140 transactions suffices to maintain this effect throughout the testing period.

4.2.2 Economic Evaluation

Baseline cost. To translate previous gas values to actual costs, we go over relevant blockchain data. Between November '22 and May '23, the ETH-to-USD exchange rate peaked at \$2120, and the average gas price paid by transactions in the 90th percentile (e.g., the upper 10% of transactions, with regard to gas price) did not exceed $106 \cdot 10^{-9}$ ETH per unit of gas. We use the previous values to compute worst-case costs: deploying the coinbase attack contract costs \$27.13, and a single computationally complex transaction invoking that contract costs \$5.3 if it is included in a block.

Long-term attacks. Claim 1 reasons about the worst-case cost of a long-term attack. We apply this result in Example 1 to provide a real-world estimate.

Claim 1. *Let* φ *and* ϕ *be the respective costs of deploying an attack contract and executing a single attack transaction, respectively. The worst-case cost of a ConditionalExhaust attack spanning* β *blocks and generating a load of* ρ *transactions per block is:* $\Phi \stackrel{def}{=} \varphi + (\phi\rho\beta(1-\alpha))$.

Proof. Recall that per the model given in Section 3, the creator of each block is picked in an independent and identically distributed (i.i.d.) manner, according to the distribution of stake among validators. We denote by X_i the random variable indicating whether a validator $v \notin S$ mined the i-th

block. Thus, using the notation introduced earlier in Section 4: $\forall i \in 1,\ldots,\beta : P(X_i = 1) = 1-\alpha$.

Recall that the cost of deploying the attack contract is denoted by φ, and the cost of a single attack transaction being accepted by ϕ. As the analysis is a worst-case one, using a high ϕ which is constant throughout the attack provides an upper bound for the cost of the ConditionalExhaust attack.

Denote the total expected cost of the attack by Φ. Given our goal of generating a computational load of ρ ConditionalExhaust transactions per block, at most ρ transactions can be accepted per block. We assume the worst-case: if a single attack transaction is accepted to a block, then all other attack transactions are accepted, too. If at some given block the transactions are not accepted due to censoring, then they are carried on to the next one. At worst, the attacker can re-send the same exact transactions, meaning that it can avoid creating new transactions with consecutive nonces, thereby lowering the cost of an attack. Thus, the expected cost of an attack is:

$$\Phi \stackrel{\text{def}}{=} \varphi + \mathbb{E}\left[\phi\rho X_1 + \cdots + \phi\rho X_\beta\right] = \varphi + (\phi\rho\beta(1-\alpha))$$

□

Example 1. *We previously established* $\varphi = \$27.13$ *and* $\phi = \$5.3$ *as the expected worst-case costs for a one-shot attack. Additionally, empirical data indicates that over* 53% *of blocks created since Ethereum's transition to PoS are OFAC-compliant [42], so we set:* $\alpha = 0.53$. *Given these parameters, the expected worst-case cost for an attack lasting* β *blocks and generating a load of* ρ *transactions per block is:* $27.13 + 2.491\rho\beta$. *For example, the expected worst-case cost of mounting an attack that generates a load of* $\rho = 140$ *ConditionalExhaust transactions per block over* $\beta = 1$ *block is* \$376. *In a best-case scenario where all validators are censoring (that is,* $\alpha = 1$*), then the attack's cost for any attack length boils down to the one-time cost of deploying the attack's contract. If no actor is censoring, the attack costs* \$770.

5 The MemPurge Attack

MemPurge allows an adversary to evict profitable transactions from the mempools of searchers, builders and proposers, and replace them with transactions that do not pay fees. This limits victims' choice of transactions when constructing bundles and blocks, thereby decreasing their revenue. We proceed by giving an overview of several attack variants. We then describe heuristics used by both geth and Flashbots' builder clients to validate mempool transactions, and present a naïve eviction strategy. This is followed by a technical description of the MemPurge attack, and an analysis of the attack.

MemPurge variant for proposers. We begin by describing an attack that serves to lead us towards a more interesting variant. If the adversary is the upcoming block proposer, it can attack any blockchain actor that uses a use a pre-funded account to spam the network with valid transaction that have consecutive nonces and thus form a "chain". If the fees offered by the transactions are high, victims will be compelled to discard existing transactions from their mempools to make room for the supposedly profitable attack transactions. These transactions can be invalidated by the attacker, by proposing a block where the block's first transaction transfers all the pre-funded account's funds to a different address. We proceed with a variant for weaker attackers.

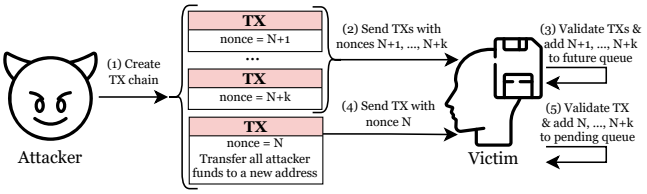

Figure 3: The MemPurge attack lowers the cost to evict transactions from victims' mempools.

MemPurge variant for non-proposers. We now describe a variant that allows an adversary who is not a proposer to attack other blockchain actors, including proposers. As before, this attack entails creating "chains" of transactions equipped with consecutive nonces, but innovates by crafting the chain to limit the number of transactions that can be incorporated in a given block, thereby reducing the cost of the attack. In particular, Ethereum recently experienced similar attacks and implemented mitigations that prevent them [43], meaning that our attack should circumvent these mitigations to succeed. The first transaction of a chain transfers all attacker funds to another account, and the rest each transfers 0 funds. These are then broadcast in the "wrong" order: the 0 value transactions are sent *first*, with the single remaining transaction sent only afterward, thereby evading the protections used by geth. A graphical summary of the attack is given in Fig. 3.

5.1 Mempool Validation

The mempool of a blockchain node is a transient database used to store candidate transactions that can be included in upcoming blocks. Due to its limited capacity and the potential impact of its contents on profits, nodes typically employ a mempool *policy* that attempts to choose transactions that increase revenue, while avoiding invalid ones.

The difficulty of ensuring transaction validity. The validity of a transaction may depend on the blockchain's state, and thus also on the transactions preceding it. E.g., a transaction transferring a positive value by user u who has 0 funds is invalid, as the user's balance cannot cover the transfer amount.

But, the transaction will be rendered valid if some preceding transaction transfers enough money to u. Thus, a single transaction may require multiple validations, for example, if the creator of the next block attempts to rearrange the block's contents to potentially capture MEV [59]. To limit the potential for DoS attacks, mempool policies, such as the one we soon describe, may use heuristics to ensure admitted transactions remain valid even when the state is slightly perturbed.

Mempool policy. Our policy, summarized in Fig. 4, follows the one used by geth and Flashbots' builder client, yet is stricter in certain cases. This makes the adversary weaker but simplifies the analysis, and ensures the attack is applicable to geth's design, as affirmed by our tests. Intuitively, the policy prioritizes high-fee transactions over low-fee ones, and pending transactions over future ones. Furthermore, if the mempool has reached its maximal capacity, then the policy prioritizes transactions sent by users with less pending transactions over those with more.

Precisely, given a mempool $\mathcal{M}$, let $|\mathcal{M}|$ be the number of transactions in $\mathcal{M}$, $\mathcal{M}^u$ be all transactions by user u in $\mathcal{M}$, and $\mathcal{M}_p, \mathcal{M}_f$ be all pending and future transactions in $\mathcal{M}$, respectively. Let the global limit on pending and future transactions be $\mu_p, \mu_f \in \mathbb{N}$, respectively, and the per-user future transaction limit be $\mu_f^u \in \mathbb{N}$. For address u, denote its balance according to the latest blockchain state by u_b. The decision to accept a transaction τ by user u into $\mathcal{M}$ proceeds as follows:

1. Reject τ if its nonce is invalid, meaning if τ_n is not larger by 1 than the nonce of u's last blockchain transaction.

2. Otherwise, reject the incoming transaction τ if the sender does not have enough funds to cover its worst-case expenses: $\sum_{\tau' \in \mathcal{M}_p^u \cup \{\tau\}} \left(\tau'_f + \tau'_v\right) > u_b$. This is a heuristic, rather than part of the consensus. It assumes each transaction always transfers its entire value, does not result in the user receiving funds from some other source (e.g., arbitrage), and consumes the gas limit in its entirety.

3. Otherwise, if the sender of τ has an existing transaction τ' in the mempool with the same nonce $\tau'_n = \tau_n$, then τ' is evicted in favor of τ if the new transaction's fee τ_f is larger than the existing transaction's fee τ'_f by at least the node's "fee bump" factor $x \geq 1$, meaning: $\tau_f \geq x \cdot \tau'_f$. If $\tau'_n = \tau_n$ and $\tau_f < x \cdot \tau'_f$, then τ is discarded.

4. Otherwise, if there is a "nonce gap" between τ and all other transactions by u, then it is wasteful to accept τ into the mempool's pending queue before the gap is filled. Precisely, if $\forall \tau' \in \mathcal{M}^u : \tau'_n + 1 < \tau_n$, then jump to step 8.

5. Otherwise, if the pending queue of the mempool has not reached its capacity (i.e., $|\mathcal{M}_p| < \mu_p$), then the incoming transaction τ is accepted to the pending queue $\mathcal{M}_p$.

6. Otherwise, the pending queue has reached its capacity. In this case, users with less pending transactions are prioritized over others. If the incoming transaction was sent by a user that has more than one pending transaction less than others in $\mathcal{M}_p$, meaning there is at least one u' such that $|\mathcal{M}_p^{u'}| > |\mathcal{M}_p^u| + 1$, then the highest-nonce transaction of u' is evicted and inserted to the future queue using rule 8, while τ takes its place. If there are several such u', τ', then one combination is chosen arbitrarily.

7. Otherwise, then the user u has at most 1 transaction less than others in the mempool. If there is another user u' that has exactly 1 transaction more than u ($\exists u' : |\mathcal{M}_p^{u'}| = |\mathcal{M}_p^u| + 1$) and has a transaction τ' with a lower fee than τ ($\exists \tau' \in \mathcal{M}_p^{u'} : \tau'_f < \tau_f$), then τ' is evicted from the pending mempool and inserted to the future section with rule 8, while τ enters instead. As before, if there are several possible u' and τ', these are chosen arbitrarily.

8. Otherwise, if the future queue has room ($|\mathcal{M}_f| < \mu_f$) and the sender of the incoming transaction τ does not reach the per-user queue limit ($|\mathcal{M}_f^u| < \mu_f^u$), accept τ to $\mathcal{M}_f$.

9. Otherwise, reject the incoming transaction τ.

Remark 2. *Nodes can change the policy to their liking. For example, some may disable rules 2, 6 and 7, as they can evict transactions in a manner which does not maximize profits. We use these rules as-is, because they weaken adversaries. Furthermore, nodes may define a policy that tries to guarantee some minimal amount of space per address, or that requires some local "threshold" fee, with transactions paying less being rejected outright. Such considerations do not qualitatively change our results, rather only potentially quantitatively (e.g., shifting attack costs by the threshold amount).*

5.2 A Naïve Eviction Strategy

Prior to introducing MemPurge, we discuss a naïve approach to evict mempool transactions. As the mempool policy prioritizes better paying transactions, one can cause a victim to evict transactions by sending enough valid high-fee transactions. While this is not an attack per-se, it serves as a baseline that one can measure MemPurge against. We now describe and analyze this eviction approach. To remain in-line with the rest of the paper, the actor that triggers the eviction and the target are called the "attacker" and "victim", correspondingly.

Description. A strategic attacker possessing substantial funds can cause victims to discard honest transactions from their mempools. Let the victim's mempool be $\mathcal{M}$, and denote the highest-fee transaction in $\mathcal{M}_p$ by τ^*. If the attacker has at least μ_p addresses each containing a minimum of τ_f^* in funds and none of which have pre-existing transactions in $\mathcal{M}_p$, the

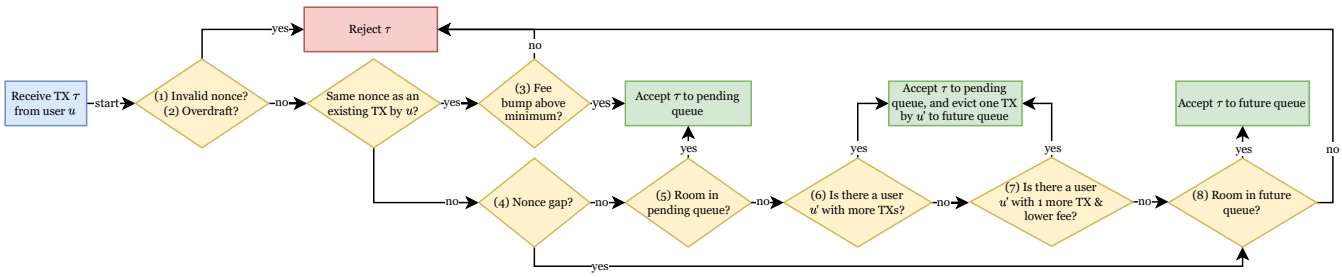

Figure 4: An overview of the mempool policy described in Section 5.1.

attacker can exploit the aforementioned mempool policy. By dispatching one transaction from each of the μ_p addresses, with every transaction paying a fee exceeding τ_f^*, the attacker can effectively evict all other transactions from the victim's mempool. An attacker wishing to evict some specific number of transactions x (not necessarily the entire mempool) can use x addresses, again sending a single transaction paying τ_f^* from each. The cost to the attacker amounts to $x \cdot \tau_f^*$.

Estimating τ_f^*. This eviction strategy succeeds if the attacker knows τ_f^*. To that end, one can employ a worst-case estimation to ensure the eviction succeeds under all circumstances, similarly to Section 4.2.2. Alternatively, an attacker that maintains a p2p connection to its victim can produce an estimation of the victim's mempool transactions, as nodes who follow the transaction gossip protocol of Section 3 both broadcast new incoming transactions and also allow peers to inquire about the presence of specific transactions. Given the parameters of Section 4.2.2, naïvely evicting all pending transactions from a mempool with a capacity of $\mu_p \stackrel{\text{def}}{=} 5120$ pending transactions (geth's default [32]), costs $24,161.

5.3 Attack Description

We present an algorithmic description of MemPurge. Intuitively, MemPurge "peels" away transactions from the mempool: at each step, the algorithm examines the highest-nonce transactions currently available, and evicts the lowest-paying one among these. The algorithm is not necessarily cost-optimal, but outperforms a naïve eviction strategy in reasonable cases. We note that the attack relies on standard value transfer transactions, without involving smart contracts.

Input. Assume the attacker wishes to evict m transactions from a victim's mempool $\mathcal{M}$, and that the attacker has a set of pre-funded accounts $\mathcal{A}^0, \mathcal{A}^1, \mathcal{A}^2, \ldots$. For simplicity, we assume the accounts have nonces equal to 0.

Output. The attack outputs: *1.* MemPurge transaction chains $\tau^{1,1}, \tau^{1,2}, \ldots, \tau^{2,1}, \tau^{2,2}, \ldots$, *2.* the number of necessary attacker accounts A, and *3.* the funds that the j-th account requires a^j, in order to execute the attack.

Initialization. Let $u^0, u^1, \ldots, u^n$ be all users with at least one transaction in $\mathcal{M}_p$, sorted in ascending order by the number of transactions they sent (u^0 has the fewest transactions, whereas u^n holds the most). For each $u \in [n]$, let $\tau^{u,j}$ be u's j-th mempool transaction by nonce order. We define the set of j-th transactions in $\mathcal{M}_p$ for all users as $N_j \stackrel{\text{def}}{=} \{\tau^{u,j} \mid \tau^{u,j} \in \mathcal{M}_p\}$, and let n^* be the length of the longest honest pending chain.

Algorithm step. At each step, a new chain is created. Intuitively, each chain is constructed and eventually broadcast to the network in a manner which prevents fees being charged from any transaction that is not the first of the chain.

Step initialization. At the beginning of a step, if m transactions or more were evicted, the attack ends. Otherwise, the account number variable is updated: $A \leftarrow A+1$, and the account's necessary pre-funded balance is initialized: $a^A \leftarrow 0$.

Create chain, part 1: set nonces and fees. For each $k = 1, \ldots, \mu_f + 1$, the chain's k-th transaction $\tau^{A,k}$ has a nonce equal to the current index: $\tau_n^{A,k} \leftarrow k$, and pays a fee higher by one: $\tau_f^{A,k} \leftarrow 1 + \min_{\tau' \in N_{n^*}} \tau'_f$, with the fee accounted for in the corresponding variable: $a^A \leftarrow a^A + \tau'_f$. Furthermore, τ' is removed from the current set: $N_{n^*} \leftarrow N_{n^*} \setminus \{\tau'\}$, and if the set is now empty, then the highest nonce is decreased: $n^* \leftarrow n^* - 1$. If $n^* < k$, then the current MemPurge chain should end with this transaction, as the chain's current length exceeds the length of the longest honest pending transaction chain (recall rules 6 and 7 of the mempool policy).

Create chain, part 2: set values. If $k > 1$, then the transaction's value is zero: $\tau_v^{A,k} \leftarrow 0$. The value of the first transaction is transferred to the address $\mathcal{A}_0$, and set to be the current account's balance, minus the transaction's fee: $\tau_v^{A,1} \leftarrow a^A - \tau_f^{A,1}$.

Finalization. After the algorithm ends, for each $j = 1, \ldots, A$, the j-th address sends transactions $\tau^{j,2}, \ldots, \tau^{j,\mu_f+1}$ to the victim, and only afterward broadcasts $\tau^{j,1}$.

Correctness. MemPurge's success in attacking geth nodes was affirmed by our testing framework in a variety of scenarios. Concretely, due to the mempool's policy, our construction allows an attacker to create a chain of overdraft transactions, yet evade being flagged for spending more funds than its balance contains. Geth performs overdraft validation when receiving transactions from users who already have pending transactions in the mempool [31]. But, per our construction, the lowest-nonce transaction of each MemPurge chain is sent *last*. This means that the other transactions from the same chain, which are received before the lowest-nonce one, are considered "future" transactions by the victim, rather than pending ones. Furthermore, when the first transaction is finally sent, geth's validation logic does not verify all the user's transactions; rather, only partial checks are done, allowing the entire chain to be considered as pending.

5.4 Evaluation

The attack's cost can be computed by running the attack's algorithm. As MemPurge is sensitive to mempool conditions, a closed-form representation is involved. Instead, we analyze a best-case scenario, followed by an empirical evaluation.

Best-case scenario. Consider an extreme hypothetical scenario where a mempool, operating under geth's default settings (mempool size μ_p of 5120 and a maximum of 64 future transactions μ_f^u per user), is completely filled with transactions exclusively from a single user. In this case, an adversary could establish 79 addresses, sending a chain of 64 transactions from each. This results in the eviction of all but $64 = 5120 - 79 \cdot 64$ victim transactions. Consequently, the adversary pays for one transaction per chain, so only 79 transactions will be paid for, considerably lower than the $5120 - 64 = 5056$ transactions required by an equivalent naïve eviction strategy.

Data. We modify geth to store all transactions received on the p2p network layer between April 18th, '23 and April 25th, '23, corresponding to blocks 17,076,370 to 17,121,301 of the Ethereum blockchain. We limit the node to at most 1,000 connections with other Ethereum peers instead of the default 50 peers, with all other parameters set to their default values. Intuitively, the number of transactions a node can observe increases with the number of peer connections. In total, we capture 6,760,060 transactions in the examined timeframe.

Fig. 5 presents a boxplot depicting the estimated per-block average mempool view, based on the distribution of unique nonce transactions per account over the examined period, for a mempool with a maximal capacity of 5120 transactions. The majority of accounts (4175.14 ± 677.01) only have one transaction. The number of addresses with 10 or more transactions drastically decreases to an average of 1.0 ± 3.0.

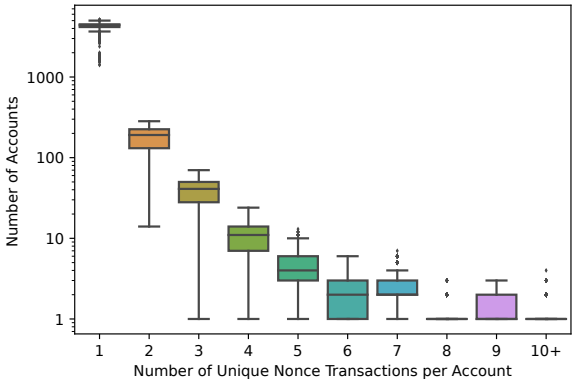

Figure 5: Boxplot depicting the estimated mempool view given a maximal capacity of 5120 transactions, based on the distribution of unique nonce transactions per account between Ethereum blocks 17,076,370 and 17,121,301 (8 days).

Empirical evaluation. Fig. 5 provides insights into the potential impact of the attack in the context of a single chain of adversarial transactions. On average, 21.43 ± 11.09 transactions can be evicted, having an average fee equal to 0.87 ± 2.16 ETH, when assuming they consume the entirety of their gas limit. We note that this is an upper bound on potential losses that can be inflicted on a victim.

5.5 ConditionalExhaust With MemPurge

Description. MemPurge can be combined with ConditionalExhaust by setting the "to" address of each MemPurge transaction to a modified ConditionalExhaust contract. The blockheight variant of the attack is implemented in Listing 2, while the full version of the paper [57] provides an implementation of a coinbase variant. Briefly, the contract changes the "simple" branch of a standard ConditionalExhaust contract to transfer all received funds to some address, thereby allowing each transaction to also implement the basic functionality of the first MemPurge transaction.

Combined attack's properties. Each chain of the combined attack consists of multiple transactions: a single computationally complex transaction, and other transactions that serve only to occupy mempool space. As these trailing transactions become invalidated by the first transaction, they are never executed and do not incur costs, similarly to MemPurge. On the other hand, as the first transaction will only be included in a block by a non-censoring actor, trailing transactions potentially reside in the mempool for a longer time, if censorship is prevalent in the network. Thus, conceptually, the combined attack preserves the good properties of the two attacks, thereby allowing an attacker to computationally

```
pragma solidity >=0.7.0 <0.9.0;
contract CombinedAttackBlockheightVariant {
 /// @notice Call this function to execute the attack.
 /// @param endDoS The end of the block range for the attack.
 function attack(uint32 endDoS) external payable {
  // Check if the current validator should be DoSed
  assembly {
   if lt(number(), endDoS) {
    let i := 565247
    for { } gt(i, 0) { i := sub(i, 1) } {
     pop(extcodehash(xor(blockhash(number()), gas())))
    }
    // Replace "CensoredAddress" with your favorite
    // sanctioned address!
    pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))
    stop()
   }
   // Replace "NextAddress" with the attacker's
   // next address
   pop(call(gas(), NextAddress, callvalue(), 0, 0, 0, 0))
   stop()
  }
 }
}
```

Listing 2: An implementation of the blockheight-variant of the ConditionalExhaust + MemPurge combined attack.

exhaust a victim while DoSing its mempool, and can also preemptively thwart potential mitigations (see Section 8).

Evaluation. We ran the combined attack through the same tests used to verify the separate attacks, and indeed the combination performs as expected when executed on a local private testnet. The gas required for deploying the coinbase variant of the attack is 131,100 and for one attack transaction is 23,711, an increase of 8.5% in the former and a negligible increase in the latter compared to the standalone ConditionalExhaust attack. The corresponding numbers for the blockheight variant are 104,769 and 21,536, again similarly increasing by 7% for deployment, and negligibly for one transaction.

6 The GhostTX Attack

The GhostTX attack allows an adversary to attack searchers by lowering their reputation in Flashbots' PBS implementation. Flashbots use reputation to prioritize actors' access to their ecosystem. A searcher's reputation is a function of its historical performance, which is measured according to the revenue per unit of gas it generated for proposers. Reputation is tied to an address, implying that a compromised searcher must rebuild its reputation from scratch using a new address. Furthermore, the attack may harm the efficient functioning of the PBS ecosystem and reduce the profits of involved builders and proposers, if high-revenue searchers are demoted. We continue by providing an overview of multiple attack variants. This is followed by a description of the necessity for reputation mechanisms in today's PBS ecosystem, and then by an implementation and evaluation of the attack.

Proposer variant. If the attacker is the proposer for the upcoming block, then it can spam searchers with "bait" transactions that appear attractive per the reputation mechanism used, yet actually harm reputation. Intuitively, under Flashbots' mechanism (which we formally define soon, in Eq. (1)), transactions that pay a high fee while consuming a low amount of gas can increase an actor's reputation if they are included in its bundles, while transactions that are never included in a block lower it. Thus, an attacker should send a "chain" of valid consecutive-nonce bait transactions, all of which pay a high amount of fee per unit of gas. Then, the attacker can invalidate all of them in one fell swoop by including a single transaction at the beginning of the upcoming block that transfers all funds from the associated address, to another one. As before, we turn our efforts to a more difficult variant.

Non-proposer variant. This variant is similar to the previous one, but requires that the adversary send transactions that conflict with bait transactions. This is because transactions sent by the adversary may propagate through the p2p network, and therefore can wind up on-chain. Per Flashbots' reputation mechanism, included transactions count towards a searcher's reputation. To not benefit its target, a conflicting transaction should be sent to other actors at the same time as a corresponding bait transaction, with both having the same nonce and the same fee. If the conflicting transaction's fee is high enough, it will be included in a block and not the bait.

6.1 Reputation Mechanisms

Flashbots' reputation. Intuitively, Flashbots' reputation score measures the average profits per gas unit produced by a given searcher. Formally, denote the set of transactions searcher U sent to Flashbots by S_U, and the subset of S_U that was included in blocks by H_U. Given a transaction T, denote its fee per unit of gas by p_T, its total gas consumption by g_T, and its payment to block builders by Δ_T. Under these notations, the reputation score r is defined in Eq. (1) [25].

$$r(U) = \frac{\sum_{T \in H_U} \Delta_T + p_T g_T}{\sum_{T \in S_U} g_T} \tag{1}$$

The necessity of reputation mechanisms. Empirical data shows Flashbots' in-house builder enjoyed an average market share of 16.8% between April and May '23 [51], implying that their reputation mechanism is important to the Ethereum ecosystem. Although we do not have evidence that others use such mechanisms, Blocknative (which operate both a builder and a relay) claim that most builders have a reputation mechanism [3], and that "the best practice for increasing searcher reputation amongst builders is to submit bundles that consistently land on-chain". Indeed, such a mechanism is important: builders that do not account for reputation based on transactions eventually entering the blockchain are exposed

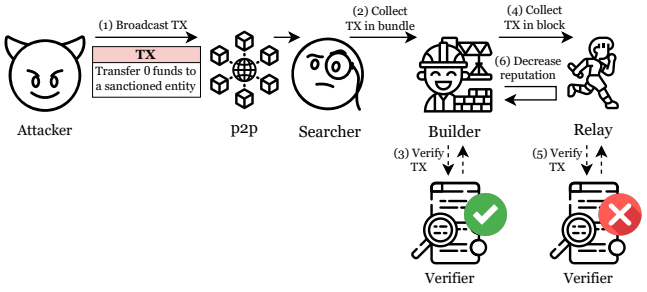

Figure 6: GhostTX's censorship variant exploits an inconsistency between builder and relay censorship methods.

```
pragma solidity >=0.7.0 <0.9.0;
contract GhostTX {
  // Replace "CensoredAddress" with a sanctioned address
  fallback () external payable {
    assembly{pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))}
  }
}
```

Listing 3: An implementation of the censorship variant of the GhostTX attack.

to trivial DoS attacks by adversarial searchers that send many transactions that impose work on victim builders yet never enter blocks, for example due to not paying high enough fees.

6.2 Censorship Variant

GhostTX's non-proposer variant can be strengthened by exploiting a discrepancy in the censorship validation functionality implemented by Flashbots' builder client, and the equivalent validation functionality that is used by Flashbots' relays. The resulting censorship variant of GhostTX is depicted in Fig. 6. An implementation for the censorship variant of GhostTX is given in Listing 3.

Censorship discrepancy. The verification function employed internally by Flashbots' builder client safeguards against the inclusion of non-compliant transactions in blocks by executing each transaction, and checking if the balances of black-listed addresses change in the interim. On the other hand, the same client exposes a verification application programming interface (API), which is primarily intended to be utilized by relay operators for validating incoming blocks sent to them by builders [22]. The API allows them to ascertain whether blocks are compliant, and it does so by executing a block in its entirety, and making sure that all involved addresses are not black-listed. Upon a detailed examination, it becomes evident that the internal function does not consider *zero fund transfers* to sanctioned entities as warranting censorship if the transfers are performed using the Ethereum virtual machine (EVM)'s *call* opcode, whereas the API does classify the same transfer as non-compliant.

Exploiting the discrepancy. An attacker can exploit the discrepancy by generating transactions that transfer 0 funds to sanctioned addresses, thereby escaping builders' internal censorship checks, while still being detected by the external API. An implementation of such a transfer is given in Listing 3. By disseminating these transactions to a multitude of searchers and builders and attaching an attractive fee to them, the adversary can ensure that these transactions are incorporated into blocks assembled by builders. However, censoring relays that receive these blocks will identify them as non-compliant, subsequently withholding them from proposers, and harming the reputation of searchers that included them in bundles. The attack is depicted in Fig. 6.

As builders unknowingly construct blocks which will be flagged as non-compliant by relays, the efforts of all involved actors are consumed in creating and verifying blocks that are ultimately discarded, wasting resources that could be employed to process legitimate transactions, and losing out on potential profits until the attack is discovered.

Correctness. We verify the attack's correctness using our testing framework, which sets up a builder node and sends it GhostTX transactions. Our tests show that attack transactions are indeed considered valid by a builder's local verification and are added to blocks, but are flagged by the API. In contrast, equivalent transactions that transfer at least 1 wei are detected by the local verification and omitted from blocks.

6.3 Evaluation

To gain a deeper insight into the efficacy of GhostTX, we collect data on the searchers involved in Flashbots' PBS ecosystem, and evaluate the attack's effect on their reputation, as determined by Flashbots' reputation system. Our evaluation intimates that launching an attack against a well-established searcher proves to be financially prohibitive. Consequently, GhostTX demonstrates greater applicability towards starting searchers, or those of average and lower performance.

Data. We compile all searcher bundles sent to Flashbots between February '21 and May '23, which were eventually included in an on-chain block, comprising 5,281,809 bundles and 8,036,039 transactions. Given the inaccessibility of bundles that were not included in blocks, we assume that searchers enjoy a success rate of 100%, meaning that $S_U = H_U$, thereby maximizing Eq. (1). Fig. 7 depicts the reputation distribution of searchers included in the data set.

Worst-case analysis: attacking the top searcher. According to our dataset, the most successful searcher made 8,240.09 ETH in profits and expended 11.26B units of gas. Assuming

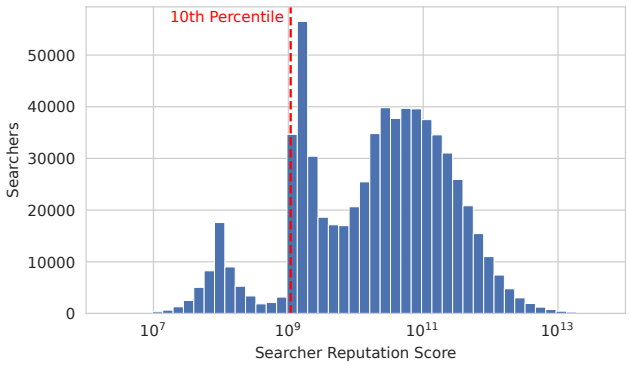

Figure 7: The reputation score distribution of Flashbots searchers, assuming a 100% success rate for each searcher.

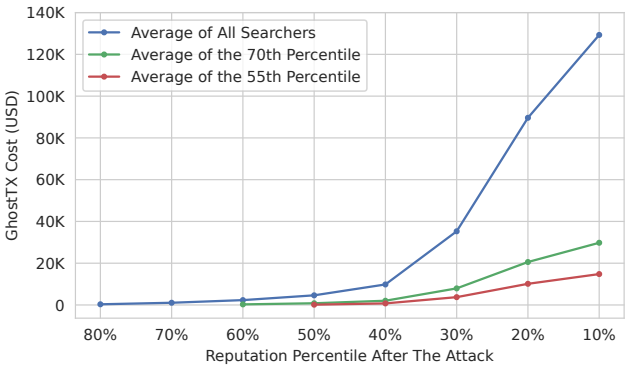

Figure 8: Cost of attacking average searchers with GhostTX.

a gas price of $106 \cdot 10^{-9}$ ETH and an exchange-rate of 2,120 USD per ETH, a GhostTX attack to displace this searcher from the upper 50% echelon of searchers costs 42.49M USD.

Attacking an average searcher. We demonstrate the applicability of GhostTX to the "average" searcher, when considering the average accumulated payment and gas expenditure over the entire data set. These average parameters are equal to a payment of 0.95 ETH and a gas consumption of 3.28M, which result in a reputation score of 2.9×10^{11}. This puts the average searcher in the 86% percentile, meaning it has a reputation that is better than 86% of all searchers. Fig. 8 elucidates the requisite USD cost to reposition this searcher across varying rank strata. Our findings suggest that an expenditure of 9.82K USD is necessitated to relegate the searcher to have a reputation that is lower than 60% of the other searchers.

Furthermore, to understand the influence of ETH payments on the cost of GhostTX, we evaluate an attack targeting searchers with a fixed reputation score of 2.9×10^{11}, and measure the attack's cost when considering different ETH payments. The results are presented in Fig. 9.

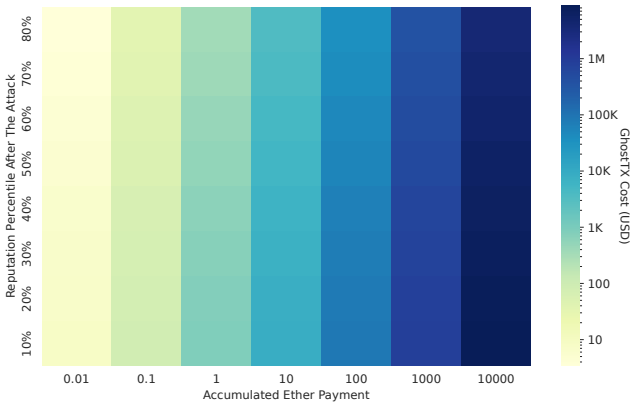

Figure 9: GhostTX cost against searchers with a fixed reputation score of 2.9×10^{11}, but with different ETH payments.

7 Practical Issues

Network-layer costs. Like previous works [44, 46, 47], our analysis does not account for potential network-layer costs. For example, the number of transactions required by our attacks may depend on their intended victims, e.g., although the time to generate 3,400 ConditionalExhaust transactions is the same as verifying one transaction on the same hardware, it may be challenging to broadcast all transactions quickly. This issue is alleviated by common services that allow users to schedule transaction to specific future blocks in advance [23]. We elaborate on these services as a separate issue.

Public transaction broadcast. We assume victims must broadcast all transactions to the network, thus increasing attack costs. This assumption may be relaxed. Services allow users to schedule *private* transactions to ecosystem actors of their choice at specific block heights, together with the promise that these will not be sent to other actors or at different times [1, 5, 23]. Thus, an adversary can privately send its transactions and schedule them to blocks corresponding to censoring validators. As private transactions are not propagated to the network, an attacker can target specific victims and be assured that its transactions will only be received by validators who cannot include them in blocks, meaning that no transaction fees will be charged for them.

Sponsored transactions An important real-world consideration that cheapens our attacks yet was not included in our analyses is the possibility of attackers using *sponsored transactions* that do not commit to their fee, instead transferring a portion of their state-dependent profits to block builders

and proposers. Such transactions are advertised by Flashbots as beneficial to actors in the MEV ecosystem, like arbitrageurs [17], and are supported by Flashbots [17], and under the name of "gas-less transactions" by Builder0x69 [1], who together captured 45% of the Ethereum builder market between April and May '23 [51]. Sponsored transactions cheapen our attacks by allowing adversaries to pay fees which ensure their transactions are only briefly viable for inclusion, thus reducing the risk of incurring losses. E.g., in Ethereum, fees can be set to slightly less than the average *base* fee, which acts as the threshold price for transaction inclusion [27].

Victim hardware. ConditionalExhaust is sensitive to victim hardware: more transactions may be needed to effectively keep stronger victims occupied. We note the inherent relationship between the victim's hardware and the number of transactions required to achieve the same effect. In particular, attacking a node using Intel i7-11370 with 4 cores, 8 threads, and 64GB RAM, an attack consisting of 80 transactions completely inhibits honest transaction inclusion in blocks, when allowing geth to use 8 threads for the block creation process. By allowing geth to use 128 threads for the same task on the 128-thread CPU of Section 4.2, an attack that similarly results in victims creating empty blocks requires 140 transactions. We note that while the number of threads is increased by a factor of 16×, the amount of transactions required for an attack of the same magnitude is only larger by a factor of 1.75×.

Load balancing. At best, load-balancing techniques have the same impact on ConditionalExhaust as increasing the thread count. Practically, if load is split among workers and then the results are combined, this opens a DoS attack vector due to interdependent transactions that invalidate each other. Indeed, a recent study by Heimbach *et al* [34] shows that, on average, Ethereum blocks contain transactions that have 4000 interdependencies. The authors find that given typical workloads, speedups achieved from such techniques cannot realistically exceed a factor of 5×.

Consensus agnosticism. Although ConditionalExhaust attack variants use PoS-specific terminology, they apply as-is to PoW blockchains. In particular, the coinbase variant of the attack does not rely on knowing the identities of other future block proposers in advance. Such knowledge of the future lowers attack costs by allowing an attacker to only target epochs with a high percentage of censoring validators. Thus, Ethereum's PoS strengthens the attack, as its leader election mechanism specifies a public leader schedule. An attacker does not have to participate in the consensus mechanism to gain this knowledge: some services provide an API endpoint for querying the upcoming validator schedule [24].

8 Mitigations

Addressing the vulnerabilities exploited by the discussed attacks is crucial for ensuring the security and integrity of the Ethereum network. We now propose potential mitigation strategies and examine their respective limitations.

Strict access lists. The censorship variant of ConditionalExhaust relies on nodes having a certain local notion of transaction validity, as dependent on their compliance with some censorship policy, with this notion not being easy to verify without executing the transaction. To allow easily verifying local policies, we suggest allowing transactions to specify *strict* access lists, that detail all addresses that they interact with, where the first non-conforming access results in a transaction reverting, with fees up to this point paid in full [2]. This allows proposers and builders to quickly verify the compliance of transactions, and provides an "insurance" that even if transactions do not conform with their lists, builders, and proposers can still receive their due compensation for executing them. Note that Ethereum allows transactions to specify *optional* access lists, where accessing an address not included in a list is penalized by higher fees [7]. These lists are not widely used and can result in higher costs [7,29,34,35]. Strict lists exacerbate the limitations of optional lists, and create new risks. Thus, if a state-dependent transaction's list does not fully account for all possible states, it may revert. Indeed, creating lists accurately is hard [35], while longer lists result in higher fees. We note that costs can be reduced by allowing contracts to have "embedded" access lists, which can apply to functions that have a well-defined execution path.

Random transaction selection. ConditionalExhaust slows down block construction because the default "greedy" transaction selection algorithm chooses the attacker's transactions first, as they have high fees [28]. If nodes would choose transactions randomly, an attacker would be required to create many more transactions to achieve the same effect. But, these transactions have lower fees, thereby harming revenue. Even when ignoring fees, we emphasize that by combining ConditionalExhaust and MemPurge, the effectiveness of this mitigation is reduced: the attack evicts honest transactions from victims' mempools, meaning that attack transactions have a greater chance of being chosen.

Limit per-account mempool slots. The MemPurge attack arises due to the ability of a single address to occupy multiple mempool slots, while paying for just a single slot per transaction chain. Such foul-play can be curtailed if mempools prohibit assigning more than a single slot per address, thereby limiting the ability of a user to create transactions that invalidate each. This means that upon receiving a transaction, if the sender's balance is higher than the transaction's total cost, the

receiving actor can be assured that no other mempool transaction can invalidate it. Yet, this mitigation is problematic for various reasons. *1.* It harms actor revenue, e.g., block builders have fewer transactions to pack into blocks. *2.* Users cannot have multiple transactions "in-flight" at the same time without resorting to costly alternatives, such as opening several accounts, or using fee-bumping to replace pending transactions with others that perform more operations, thereby hampering user experience. *3.* The mitigation is partial, it does not prevent the proposer and censorship variants of the attack.

GhostTX. GhostTX's non-proposer variant can be made harder to execute by ensuring Flashbots' validity checks are identical across all implementations. This does not prevent all attacks: the validation discrepancy only gives adversaries more time to propagate conflicting GhostTX transactions, and the non-censorship variants do not rely on it.

Proposer variants. Although outside our model, adversarial block proposers were briefly mentioned to show that if adversaries know in advance when they will be elected to propose blocks, they can cheaply execute attacks. A potential mitigation is to use mechanisms where leaders have only probabilistic knowledge of future roles, such as PoW. Without this foresight, being a proposer would only confer some probabilistic advantage when it comes to our attacks, thereby increasing potential associated costs. This is a novel observation: the literature has so far focused on making the identity of future leaders private from other actors to protect leaders from attacks, whereas our attacks show that the identity of a leader should also be hidden from the leader itself.

9 Related Work

REAs. This genre of blockchain attacks was inaugurated by the "Broken Metre" attack of Perez & Livshits [47], which is designed to exhaust victim resources, primarily CPU and IO. The authors used a genetic algorithm to craft adversarial transactions that maximize resource usage, while minimizing the fees incurred for computational load by relying on EVM opcodes were mispriced relative to their resource use. The latter is of significance, as the work assumed that attack transactions will enter the blockchain, thereby also requiring adversaries to cover their gas costs. The cost of the offending opcodes was corrected in 2021 [50], thereby partially mitigating the attack by increasing its cost.

ConditionalExhaust instead minimizes attack costs by relying on two execution branches, where the first is computationally demanding yet is only triggered when executed by those who cannot include it in blocks, and the second is cheap. We compare a single ConditionalExhaust transaction to an equivalent "Broken Metre" transaction, where both consume a block's entire gas quota. Using the parameters of Section 4.2.2, one ConditionalExhaust transaction costs $5.3, and a "Broken Metre" transaction costs $6741.

Soft-fork DAO DoS. In 2016, an Ethereum contract called "The DAO" was hacked by adversaries who transferred funds then worth $53 million to a contract called "The Dark DAO" [36]. The so-called DAO soft-fork proposal suggested preventing the adversaries from using these funds by requiring all Ethereum actors to consider transactions interacting with The Dark DAO as invalid [36]. Hess *et al.* present an attack on the proposal in a blog post, where adversaries DoS victims by sending transactions that interact with The Dark DAO [36]. The soft fork was abandoned for a proposal that is not susceptible to the attack [9].

In contrast, ConditionalExhaust is applicable to Ethereum. While the DAO DoS targets "global" censorship practices adopted by all blockchain actors and could be mitigated by charging fees from non-compliant transactions, the censorship variant of ConditionalExhaust cannot be mitigated as it targets "local" censorship practices that are not enforced by consensus, such as compliance with OFAC's regulations. This difference is important, as it implies that attack transactions may incur fees from adversaries if they are eventually added to blocks by non-compliant actors. To that end, we design transactions that are complex for compliant actors, yet simple for non-compliant ones, and thus cheap if added by the latter to blocks. Furthermore, we note that by combining ConditionalExhaust and MemPurge, one obtains a stronger attack that both exhausts computational and mempool resources.

Mempool DoS attacks. Li *et al.* [44] conceived the category of mempool DoS attacks, with their DETER attacks. These allow adversaries to evict mempool transactions by creating low-fee transactions. The vulnerabilities exploited by their attacks were mitigated in geth version 1.11.4, released on March '23 [30, 43]. Prior to these mitigations, the authors exploited geth's mempool policy in two attacks.

DETER-X exploits the possibility of the unmitigated policy evicting low-fee pending transactions for high-fee future one. The attack spams the network with high-fee future transactions that have a nonce gap which is never filled, prompting victims to evict valuable pending transactions for them. This attack was mitigated by ensuring that the policy never evicts pending transactions for future transactions (policy rule 4).

DETER-Z exploits the unmitigated policy's isolated validation of transactions: incoming transactions are validated without considering previous pending transactions sent by their senders. The attack sends chains of transactions where each drains the attacker's funds. Thus, a chain's first transaction is valid, and the rest are not. The attack was mitigated by validating new transactions with their senders' existing pending mempool transactions, and ensuring their total worst-case costs do not exceed the senders' balances (policy rule 2).

MemPurge works on patched versions of geth, and evades the mitigations by employing a multiphase approach and sending transactions out-of-order. Moreover, the mitigations and rules 5, 8 of the mempool policy force adversaries to pay for more transactions. Thus, MemPurge constructs attack chains in a manner that lowers costs.

10 Conclusion

This study brings to light the consequences and security challenges of speculative transaction execution in expressive smart contract blockchains. By proposing and evaluating the ConditionalExhaust, MemPurge, and GhostTX attacks, we uncover critical vulnerabilities within Ethereum's ecosystem that malicious actors may exploit.

Reproduction

Reproduction details can be found in the full version of the paper [57].

Acknowledgements

This work was partially supported by the Ministry of Science & Technology, Israel, and by a grant from the Ethereum Foundation (EF). We would like to extend our gratitude to our reviewers for their insightful feedback, which served to improve the work considerably. We furthermore would like to thank the EF and the Flashbots company for the prizes they have awarded the authors for the findings made in this paper.

References

[1] beaverbuild. Rpc docs, 2024. URL: https://beaverbuild.org/docs.html.

[2] Alex Beregszaszi and Nikolai Mushegian. Eip-140: Revert instruction, 2017. URL: https://eips.ethereum.org/EIPS/eip-140.

[3] Blocknative. Mev bundle failure: Troubleshooting why your bundle didn't end up on-chain, January 2023. URL: https://www.blocknative.com/blog/mev-bundle-failure.

[4] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE symposium on security and privacy*, pages 104–121, San Jose, CA, USA, 5 2015. IEEE, IEEE. doi:10.1109/SP.2015.14.

[5] Builder0x69. Builder0x69 json-rpc api documentation, 2023. URL: https://web.archive.org/web/20230928131626/https://docs.builder0x69.io/.

[6] Benedikt Bünz, Steven Goldfeder, and Joseph Bonneau. Proofs-of-delay and randomness beacons in ethereum, 2017.

[7] Martin Buterin, Vitalik; Swende. Eip-2930: Optional access lists, August 2020. URL: https://web.archive.org/web/20230616054341/https://eips.ethereum.org/EIPS/eip-2930.

[8] Vitalik Buterin. Geth nodes under attack again; we are actively working on it., 2016. URL: https://reddit.com/r/ethereum/comments/55s085/geth_nodes_under_attack_again_we_are_actively.

[9] Vitalik Buterin. Hard fork completed, July 2016. URL: https://web.archive.org/web/20160814023106/https://blog.ethereum.org/2016/07/20/hard-fork-completed/.

[10] Stefanos Chaliasos, Arthur Gervais, and Benjamin Livshits. A study of inline assembly in solidity smart contracts. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1123–1149, 2022. doi:10.1145/3563328.

[11] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In Joseph K. Liu and Pierangela Samarati, editors, *Information Security Practice and Experience*, pages 3–24, Cham, 2017. Springer International Publishing.

[12] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 910–927, San Francisco, CA, USA, 2020. IEEE. doi:10.1109/SP40000.2020.00040.

[13] Maya Dotan, Aviv Yaish, Hsin-Chu Yin, Eytan Tsytkin, and Aviv Zohar. The vulnerable nature of decentralized governance in defi. In *Proceedings of the 2023 Workshop on Decentralized Finance and Security*, DeFi '23, page 25–31, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3605768.3623539.

[14] ethereum. Ethereum wire protocol (eth), April 2023. URL: https://github.com/ethereum/devp2p/blob/master/caps/eth.md.

[15] Ethereum. Proposer-builder separation, May 2023. URL: https://github.com/ethereum/ethereum-org-website/blob/1729448/src/content/roadmap/pbs/index.md.

[16] ethstaker guides. Mev relay list for mainnet, 2023. URL: https://github.com/eth-educators/ethstaker-guides/blob/9bb22c64/MEV-relay-list.md.

[17] Flashbots. searcher-sponsored-tx, 2021. URL: https://github.com/flashbots/searcher-sponsored-tx.

[18] Flashbots. Introduction, 2022. URL: https://github.com/flashbots/flashbots-docs/blob/e1683f8/docs/flashbots-mev-boost/introduction.md.

[19] Flashbots. system-requirements, 2022. URL: https://web.archive.org/web/20221129203757/https://docs.flashbots.net/flashbots-mev-boost/getting-started/system-requirements.

[20] Flashbots. builder, 2023. URL: https://github.com/flashbots/builder.

[21] Flashbots. builder: Blacklisting addresses, 2023. URL: https://github.com/flashbots/builder/blob/481f1c3/README.md?plain=1#L128.

[22] Flashbots. mev-boost-relay: Builder submission validation nodes, 2023. URL: https://github.com/flashbots/mev-boost-relay/blob/171c1aa/README.md?plain=1#L201.

[23] Flashbots. Private transactions, 2023. URL: https://web.archive.org/web/20230521052523/https://docs.flashbots.net/flashbots-auction/searchers/advanced/private-transaction.

[24] Flashbots. Relay api, 2023. URL: https://web.archive.org/web/20230128125132/https://flashbots.github.io/relay-specs/.

[25] Flashbots. Searcher reputation, 2023. URL: https://web.archive.org/web/20230203073040/https://docs.flashbots.net/flashbots-auction/searchers/advanced/reputation/.

[26] Yotam Gafni and Aviv Yaish. Greedy transaction fee mechanisms for (non-)myopic miners, 2022. doi:10.48550/arXiv.2210.07793.

[27] Yotam Gafni and Aviv Yaish. Barriers to collusion-resistant transaction fee mechanisms, February 2024. doi:10.48550/arXiv.2402.08564.

[28] Yotam Gafni and Aviv Yaish. Competitive revenue extraction from time-discounted transactions in the semi-myopic regime, February 2024. doi:10.48550/arXiv.2402.08549.

[29] Matt Garnett. Eip-3521: Reduce access list cost, April 2021. URL: https://web.archive.org/web/20230329161516/https://eips.ethereum.org/EIPS/eip-3521.

[30] go ethereum. txpool2_test.go, March 2023. URL: https://github.com/MariusVanDerWijden/go-ethereum/blob/d1de0bf/core/txpool/txpool2_test.go#L146.

[31] Go-Ethereum. txpool.go.validatetx, 2023. URL: https://github.com/ethereum/go-ethereum/blob/ba09403/core/txpool/txpool.go#L677.

[32] The go-ethereum Authors. Command-line options, 2023. URL: https://web.archive.org/web/20230410005002/https://geth.ethereum.org/docs/fundamentals/command-line-options.

[33] Chris Hager. remove example blacklist, March 2023. URL: https://github.com/flashbots/builder/pull/56.

[34] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, and Roger Wattenhofer. Defi and nfts hinder blockchain scalability. In Foteini Baldimtsi and Christian Cachin, editors, *Financial Cryptography and Data Security*, pages 291–309, Cham, 2024. Springer Nature Switzerland.

[35] Lioba Heimbach, Quentin Kniep, Yann Vonlanthen, Roger Wattenhofer, and Patrick Züst. Dissecting the eip-2930 optional access lists, 2023. arXiv:2312.06574.

[36] Tjaden Hess, River Keefer, and Emin Gün Sirer. Ethereum's dao wars soft fork is a potential dos vector, June 2016. URL: https://web.archive.org/web/20230919110047/https://hackingdistributed.com/2016/06/28/ethereum-soft-fork-dos-vector/.

[37] Alejo; Hasu Hu, Elaine; Salles. The cost of resilience, November 2022. URL: https://web.archive.org/web/20230325222151/https://writings.flashbots.net/the-cost-of-resilience.

[38] Kamil Jezek. Ethereum data structures, 2021. URL: https://arxiv.org/abs/2108.05513, doi:10.48550/ARXIV.2108.05513.

[39] Paweł Johnson, Nick; Bylica. Eip-1052: Extcodehash opcode, 2018. URL: https://eips.ethereum.org/EIPS/eip-1052.

[40] Sam Kessler. Vitalik buterin's new ethereum road map takes aim at mev and censorship, November 2022. URL: https://www.coindesk.com/tech/2022/11/09/vitalik-buterins-new-ethereum-roadmap-takes-aim-at-mev-and-censorship/.

[41] Lucianna Kiffer, Asad Salman, Dave Levin, Alan Mislove, and Cristina Nita-Rotaru. Under the hood of the ethereum gossip protocol. In *International Conference on Financial Cryptography and Data Security*, pages 437–456, Berlin, Heidelberg, 2021. Springer, Springer.

[42] Labrys. Mev watch, April 2023. URL: https://web.archive.org/web/20230428094150/https://www.mevwatch.info/.

[43] Felix Lange. Release vana (v1.11.4), March 2023. URL: https://github.com/ethereum/go-ethereum/releases/tag/v1.11.4.

[44] Kai Li, Yibo Wang, and Yuzhe Tang. Deter: Denial of ethereum txpool services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 1645–1667, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3485369.

[45] Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, Zheyuan He, Renkai Jiang, Ting Chen, Jiachi Chen, Tao Zhang, and Xiaosong Zhang. Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts. *IEEE Trans. Software Eng.*, 49(2):777–801, 2023. doi:10.1109/TSE.2022.3163614.

[46] Michael Mirkin, Yan Ji, Jonathan Pang, Ariah Klages-Mundt, Ittay Eyal, and Ari Juels. Bdos: Blockchain denial-of-service. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 601–619, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372297.3417247.

[47] Daniel Perez and Benjamin Livshits. Broken metre: Attacking resource metering in EVM. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, Reston, VA, 2020. The Internet Society. URL: https://www.ndss-symposium.org/ndss-paper/broken-metre-attacking-resource-metering-in-evm/.

[48] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mLSM: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association. URL: https://www.usenix.org/conference/hotstorage18/presentation/raju.

[49] Cosimo Sguanci and Anastasios Sidiropoulos. Mass exit attacks on the lightning network, 2022. URL: https://arxiv.org/abs/2208.01908, doi:10.48550/ARXIV.2208.01908.

[50] Peter Swende, Martin Holst; Szilagyi. Dodging a bullet: Ethereum state problems, 2021. URL: https://blog.ethereum.org/2021/05/18/eth-state-problems.

[51] Titan. Builder dominance and searcher dependence, 2023. URL: https://frontier.tech/builder-dominance-and-searcher-dependence.

[52] Anton Wahrstätter, Jens Ernstberger, Aviv Yaish, Liyi Zhou, Kaihua Qin, Taro Tsuchiya, Sebastian Steinhorst, Davor Svetinovic, Nicolas Christin, Mikolaj Barczentewicz, and Arthur Gervais. Blockchain censorship, 2023. arXiv:2305.18545.

[53] Jeffrey Wilcke. The ethereum network is currently undergoing a dos attack, 2016. URL: https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/.

[54] David Yaffe-Bellany. Investors sue treasury department for blacklisting crypto platform, September 2022. URL: https://www.nytimes.com/2022/09/08/business/tornado-cash-treasury-sued.html.

[55] Aviv Yaish, Svetlana Abramova, and Rainer Böhme. Strategic vote timing in online elections with public tallies, February 2024. arXiv:2402.09776, doi:10.48550/arXiv.2402.09776.

[56] Aviv Yaish, Maya Dotan, Kaihua Qin, Aviv Zohar, and Arthur Gervais. Suboptimality in defi. Cryptology ePrint Archive, Paper 2023/892, 2023. URL: https://ia.cr/2023/892.

[57] Aviv Yaish, Kaihua Qin, Liyi Zhou, Aviv Zohar, and Arthur Gervais. Speculative denial-of-service attacks in ethereum. Cryptology ePrint Archive, Paper 2023/956, 2023. URL: https://ia.cr/2023/956.

[58] R. Yang, T. Murray, P. Rimba, and U. Parampalli. Empirically analyzing ethereum's gas mechanism. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 310–319, Los Alamitos, CA, USA, jun 2019. IEEE Computer Society. doi:10.1109/EuroSPW.2019.00041.

[59] Liyi Zhou, Kaihua Qin, and Arthur Gervais. A2MM: mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges, 2021. URL: https://arxiv.org/abs/2106.07371, arXiv:2106.07371.