Congesting Ethereum after EIP-1559

***Abstract*—We provide two novel block congestion attacks on Ethereum that are applicable even in the presence of the EIP-1559 base fee mechanism, which aimed to make such attacks impossible or highly costly. Unlike traditional block congestion methods, our approaches allow the attacker to avoid paying large transaction fees in case the attack is unsuccessful. Moreover, our second attack avoids an explosion in the block base fee and can thus be used for prolonged congestion of an interval of blocks. Finally, we provide real-world examples of contracts currently deployed on the Ethereum blockchain which are vulnerable to such attacks. Thus, block congestion is both possible and profitable, even after EIP-1559.**

I. Introduction

Gas [1]. In Ethereum, to avoid denial-of-service attacks, a user who initiates a function call transaction has to pay a gas fee which is roughly proportional to the amount of computational resources used in the execution of their desired function. Initially, Ethereum followed the first-price auction mechanism, in which each transaction specified the maximum amount g_m of gas that the initiator was willing to pay for, as well as a gas price p. If the transaction used $g \leq g_m$ units of gas, then the miner would be paid a transaction fee of $g \cdot p$. If it used more than g_m units of gas, an out-of-gas exception would be raised, its execution would stop, all its effects would be reverted and the miner would be paid $g_m \cdot p$. Each block was limited to using at most 30 million units of gas. Thus, the rational miners tended to prefer transactions with higher gas prices to optimize their earnings. This rationality could then be exploited to censor certain transactions or reorder them [2], [3]. Transaction fees can even be used to incentivize the miners to create a fork [4]. This gas model had a variety of other downsides, as well, including unpredictable transaction fees and a vulnerability to congestion attacks, i.e. when an attacker creates many transactions with slightly higher than average gas price to fill up the blocks and thus stop another user's transaction from being added to the blockchain. There are many real-world scenarios where congestion, be it an attack or unintentional, caused considerable financial losses [5], [6].

EIP-1559 [7], [8]. To address the problems above in the gas model, Ethereum Improvement Proposal (EIP) 1559 was proposed in 2019 and included in the London Hard Fork of 2021. After EIP-1559, Ethereum's transaction fee model has been significantly revamped. There is a new concept called *base fee*, which is meant to match supply and demand for block space and avoid congestion. Specifically, the gas price p is now of the form $p = b + t$, where b is the base fee of the block, which has to be paid by every transaction in this block, and t is a tip (priority fee) decided by the user. The user pays $g \cdot p$, but $g \cdot b$ units of the transaction fee are burnt and only $g \cdot t$ is paid to the miner. Moreover, the base fee is adjusted after every block. If the previous block uses a lot of gas, e.g. close to 30 million units, then the base fee increases. Conversely, if it uses significantly less than 15 million units of gas, the base fee decreases. Specifically, we have

$$b_n = b_{n-1} \cdot (1 + \frac{1}{8} \cdot \frac{g_{n-1} - g_{target}}{g_{target}}), \tag{1}$$

where b_n is the base fee of block n, b_{n-1} is the base fee of the previous block, g_{n-1} is the total gas consumed in the previous block and g_{target} is the target gas usage per block, currently set at 15 million.

Congestion Attack. Consider a smart contract in which there is a timelock. A user wants to submit a transaction to this smart contract and has to do it before a particular block number. An attacker can try to ensure that the miners do not include this user's transaction in their blocks before the deadline. This is called a congestion attack and can potentially be profitable to the attacker. For example, if the smart contract implements an auction, the attacker's aim might be to stop an honest user from placing/revealing a bid. In general, a successful congestion attack would affect the security of time-sensitive smart contracts such as on-chain voting protocols, auctions and payment channels that rely on deadlines [9]. Attacking timelocked contracts is itself an active area of research, e.g. [10] proposes a bribing attack on time-locked transactions and [11] explores a flooding attack on the Bitcoin Lightning network, which prevents the settlement of debts in the channels and steals the unlocked funds.

Congestion after EIP-1559. With the new gas model of EIP-1559, a congestion attack is still possible [12] and an attacker can keep creating many high-tip transactions to fill up the 30 million gas capacity of a block, but this is expected to be highly costly and not scale beyond a few blocks since the base fees keep increasing exponentially if the blocks use a lot of gas. On the other hand, if an attacker does not provide enough transactions to almost fill up a block, then there is no guarantee that the miners would not include the victim's transaction, too. In any case, an attacker who creates many transactions with the hope of excluding another user's transaction from a block risks a worst-case scenario in which most of his transactions are added to the block, and thus he has to pay their transaction fees, but the victim's transaction is also included. Thus, there is a risk of paying for an unsuccessful attack.

Our Contribution. In this work, we provide two novel congestion attacks on the Ethereum blockchain.

- Our first attack is a simple variant of classical congestion in which the attacker pays a high transaction fee *if and only if* his transaction is the only transaction in the block. Thus, an unsuccessful attack on a block comes with a negligible cost. Although a transaction cannot access other transactions in the same block, we can ensure this property since we can require that the gas left for our transaction is close to 30 million.
- Our second attack, which is our main technical contribution, is a further refinement in which the block base fee does *not* increase even in the presence of EIP-1559.

Thus, we can congest a long sequence of consecutive blocks without having to incur the prohibitive and exponential cost of increased base fees. This attack effectively nullifies one of the main selling points of EIP-1559.

- For both attacks, we provide game-theoretic guarantees showing that rational miners will collaborate with the attacker and allow the attack to succeed.
- Finally, we also provide examples of real-world contracts on the Ethereum blockchain which are vulnerable to these attacks. Thus, block congesting attacks on Ethereum are both possible and profitable, even after EIP-1559.

II. Gas-Hungry Transaction Attack

Our first attack is quite simple, and can be seen as an improvement of a classical block congestion attack envisaged in [12]. This is not our main contribution and we are presenting it only to set up the scene for the next attack. Our approach ensures the attacker pays only a negligible transaction fee in case of failure. The idea is to create a transaction that uses so much gas as to make it impossible for the miner to include any other transactions in the block. This will ensure effective congestion as the attacking transaction is guaranteed to be the only transaction in its block.

Suppose the attacker's goal is to congest block number x. He first deploys the smart contract in Algorithm 1 before block x. He then creates a transaction that calls the `congest`(x) function with a gas limit of 30 million and a tip price that is larger than the miner's expected average. He publishes this transaction when the time of block x comes. In Algorithm 1, c is a small number. In practice, we set $c = 300$. The function first checks that the call to `congest`(x) is the first transaction in the block by ensuring that very little gas has been spent beforehand. It then checks the block number. If both checks pass, it runs an infinite loop which uses the entire possible gas of a block and thus pays the maximum possible transaction fee to the miner. Thus, a rational miner will always include this transaction in their block. Moreover, note that no transaction can be added after the call to `congest()` since this call already exhausts all the available gas in the block. Additionally, as the transaction pays a high gas fee only if it is the first transaction of the block, a rational miner would not include any transactions before this one. This provides game-theoretic guarantees that our attack transaction would be the only transaction of block x and hence the attack succeeds.

```
Function congest(x):
    // Check if this is the first
       transaction in the block
    require gasleft() ≥ gaslimit() − $c$ and
     block.number == x
    while true do
        // gas exhaustion loop
    end
```
Algorithm 1: Gas-Hungry Attack on a Single Block

We also note that this transaction has no effect on the state/storage of the contract since either fails the require statement or runs out of gas and is thus always reverted. This function's only role is to pay a huge gas fee to the miner *if and only if* they do not include any other transactions in their block. If the miner is irrational and decides to include transactions before our attack transaction or to mine it in a block other than x, then the require statement will fail and the gas usage and transaction fee will be tiny. Thus, the attacker pays a huge transaction fee if and only if his attack is successful.

Additionally, as long as we choose the gas tip to be large enough, if a miner does not cooperate and include our transaction in their block x, this immediately creates an incentive for the next miner to fork this block out and thus win the high transaction fees of `congest()`. This is similar to an attack in [4]. Thus, as long as the miners are rational and aware of the attack, they have every incentive to cooperate with it.

Algorithm 1 shows how one can congest a particular block x. However, in most real-world cases, an attacker is interested in congesting an interval $[x_1, x_2]$ of blocks to exclude a victim's transaction to a timelocked contract. Algorithm 2 shows how the attack can be extended to an interval of blocks. Here, the attacker first deploys the contract before time x_1 and then keeps calling `congest`(x) at every block x between x_1 and x_2, while setting a tip price that is above what the miners can expect to receive from other transactions.

```
Global variable: last_congested = $x_1 - 1$;
Function congest(x):
    require gasleft() ≥ gaslimit() − $c$  and
     block.number == x;
    require block.number == last_congested + 1;
    last_congested = block.number;
    while gasleft() > $c$ do
        // Gas exhaustion loop
    end
```
Algorithm 2: Gas-Hungry Attack on an Interval of Blocks

The main changes in this variant are as follows: (i) the attacking contract keeps track of the number of the last congested block and proceeds to congest the current block only if the previous block was already congested. This ensures that we successfully congest an interval of blocks, (ii) we can no longer let the calls to `congest`() be reverted due to out-of-gas exceptions since that would revert the updates to the variable last_congested. Thus, our while loop terminates as soon as the potential remaining gas up to the block gas limit is so small that no other transaction can be added after ours.

Using the same arguments as in the case of Algorithm 1, it is easy to see that the attacker pays a large gas fee for the call `congest`(x) if and only if he successfully congests block x. Similarly, the miners are financially incentivized to cooperate with the attack. A rational miner of block $x \in [x_1, x_2]$ has no other choice for the set of transactions included in their block that would yield a higher total revenue. Also, if the miner of block x does not cooperate with the attack, the miners of blocks $[x+1, x_2]$ cannot receive high revenues either. Thus, they are incentivized to fork block x and add an alternative block x that cooperates with the attack. Overall, as long as the miners are rational and the attacker can afford the transaction fees, the attack will be successful in congesting all blocks in the interval $[x_1, x_2]$.

In the original first-price auction model of gas, this attack would not cost much. Indeed, the cost of congesting k blocks would simply be $k \times 3 \times 10^7$ units of gas. However, EIP-1559 makes the attack much costlier and thus applicable to only short intervals. Specifically, since each of our attack blocks is using almost 30 million units of gas, i.e. $2 \cdot g_{target}$, applying the formula in Equation (1) shows that the base fee is multiplied by $9/8$ after each block and thus increases exponentially. Figure 1 shows the total cost of congesting k blocks assuming that the initial base fee was 23.65 Gwei and the priority fee is 13 Gwei. On Ethereum, these are considered a usual base fee and a generous tip.

III. Verified Single Transaction Attack

To design our second attack, we first take a deeper look at the attack of the previous section. The crucial property of that attack was to ensure that the miner(s) would be paid a large amount if and only if they included the attacker's transaction, and nothing else, in their block. This incentivized them to cooperate with the attack and congest the block. We were essentially bribing the miners to do this by relying on a clever smart contract that would pay them a large amount if they cooperated. Unfortunately, the bribe was paid via gas. Thus, it led to huge blocks with gas usages that were close to 30 million units, causing an exponential growth in the base fee. Thus, the attacker had to pay an outrageously large total base fee, but this was burnt and never paid to the miners, who only received the tips. We now provide a smarter attack that verifies whether the miner cooperated with the attack and rewards them accordingly, but does this without relying on gas. Our attack achieves an ***exponential decrease*** in the base fee! Thus, the contribution of the base fee to the total attack cost becomes insignificant and we get the same level of scalability as we had for the previous attack in the absence of EIP-1559. In other words, this attack effectively circumvents EIP-1559's disincentives for congestion.

A pseudocode of our attack contract is provided in Algorithm 3. As before, assume that the attacker wishes to congest

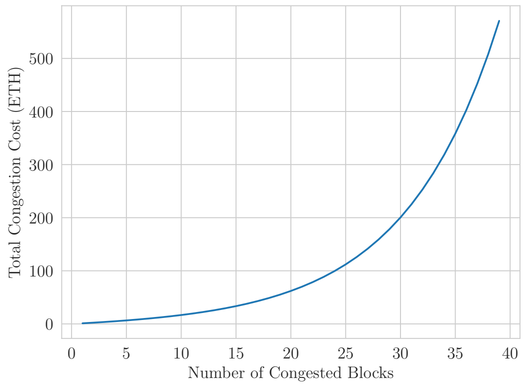

Fig. 1: Total cost of our first attack based on the number of congested blocks.

```
Global variables: $r, x_1, x_2$, last_congested = $x_1 - 1$;
Global variables: $m[], h[]$, last_paid = $x_1 - 1$;
Function congest($x$):
    require block.number == $x$;
    $m$[block.number] = block.coinbase;
    $h$[block.number] = block.hash(block.number)
Function verify($b, x$):
    require $x_1 \leq x \leq x_2$;
    require block.number $> x_2$;
    require last_congested == $x - 1$;
    require hash($b$) == $h[x]$;
    require $b$ contains only one transaction which is a
      call to congest($x$);
    last_congested = $x$;
Function payout($x$):
    require $x_1 \leq x \leq x_2$;
    require last_congested == $x_2$;
    require last_paid == $x - 1$;
    last_paid = $x$;
    $m[x]$.pay(r);
```

Algorithm 3: Verified Single Transaction Attack on an Interval of Blocks

blocks in the range $[x_1, x_2]$. He first chooses an amount r that should be paid as a reward/bribe to each of the miners. r should exceed the tip amounts that the miners expect to receive for one block. He then deploys this contract before x_1 and deposits $(x_2 - x_1 + 1) \cdot r$ in the contract. He then calls `congest`(x) for every $x \in [x_1, x_2]$. Note that `congest`(x) is a very simple function that uses a tiny amount of gas. All it does is to record the address of the miner of block x, i.e. $m[x]$, as well as the hash of the block, for future use. Ethereum guarantees that a transaction cannot read the other transactions in the same block. So, we have no direct way of checking in `congest`(x) whether the current function call is the only transaction of block x. However, we can access and record the hash of this block.

We want to make sure the miner of block $x \in [x_1, x_2]$ is incentivized to include only the transaction `congest`(x) in their block. So, we will pay them the reward x only if they can prove they cooperated. More specifically, after block x_2, the miner of each block $x \in [x_1, x_2]$ should call the function `verify`(b, x). Here, x is the block number and b is the block that this miner added to the blockchain. In other words, we are asking the miner to create a later transaction in which they pass their mined block b as a parameter to our smart contract. Since the contract had already recorded the hash of this block, it can verify that b is genuine and not tampered with. Additionally, b includes the root hash of the Merkle-Patricia tree of transactions included in block x [1]. Thus, the function `verify` can simply form the Merkle-Patricia tree containing only a single call to `congest`(x) and verify that the root of this tree is included in b. This proves that the miner of block x did not include any other transactions in their block.

Finally, if all the blocks within the range $[x_1, x_2]$ pass the verification, the miners can call the `payout` function and receive their rewards. In practice, we can set a deadline for this and then return the money to the attacker if the miners could not claim the payments by that deadline, e.g. because they did not pass the verification step. We note several desirable

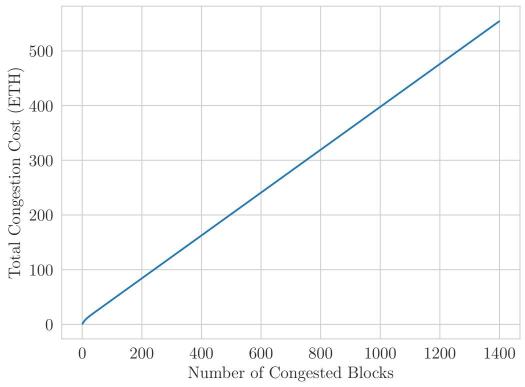

Fig. 2: Total cost of our second attack based on the number of congested blocks.

properties of our attack:

- The rewards are paid to the miners *if and only if* all the blocks in the interval $[x_1, x_2]$ are congested and the attack was totally successful. Otherwise, the attacker gets his money back and has only had to pay a negligible transaction fee for deploying the contract and the calls to `congest`.
- Each miner of a block $x \in [x_1, x_2]$ is incentivized to cooperate with the attack since the reward/bribe r exceeds any tips or rewards that they could earn from including other transactions in their block.
- As in the previous attack, if a miner does not cooperate, the other miners have an incentive to fork their block out. This is because every miner's rewards depends on the whole interval being congested.
- In each congested block $x \in [x_1, x_2]$, the only transaction is a call to `congest`(x), which uses a tiny and constant amount of gas, storing only two values in the contract's storage. Thus, the total gas usage of the block x is tiny and close to zero. Applying EIP-1559's formula in Equation (1) shows that the base fee is multiplied by $\approx 7/8$ after each congested block. Therefore, our base fees are ***decreasing exponentially*** and rapidly tending to zero if we apply the attack over a prolonged period.

Given the above, congesting k blocks with this approach costs $O(k \cdot r)$. The cost is dominated by the bribe that is paid to the miners. The transaction fees paid by the attacker are a constant amount per congested block and do not increase from one block to the next, since the base fees are now decreasing. Figure 2 shows the total cost of this attack for congesting k blocks assuming the initial base fee b was 23.65 Gwei and the tip t is 13 Gwei. We also assume that we pay a bribe of $r = 3 \times 10^7 \cdot (b + t) \times 1.002$ to the miners. This is a usual base fee and the set rewards are highly generous, surpassing anything that the miners can realistically earn by mining a different block. Compared to the previous attack, a much larger range of blocks can be congested with the same cost.

IV. Examples of Vulnerable Contracts

To find contracts that use timelocks and are potentially vulnerable to our attack, we analyzed the bytecodes of smart contracts deployed between blocks 14497034 and 18375639 on the Ethereum blockchain. Our experiments revealed 271,860 contracts that employ the `TIMESTAMP` or `NUMBER` opcodes. Among these, 16,749 were verified on EtherScan [13] and had a non-zero ETH balance. Thus, it is clear that a large number of contracts have timelocks and are potentially vulnerable. We then manually analyzed the examples with a balance of more than 25 ETH to see if a block congestion attack is really applicable. We found 7 examples of vulnerable contracts (Table I). In some cases, the attack requires a few extra steps.

TABLE I: Examples of Real-world Ethereum Contracts that are Vulnerable to our Block Congestion Attacks.

Address:	**Name:**	**ETH Balance:**	**Vulnerability:**
0xF42c318dbfBaab0EEE040279C6a2588Fa01a961d	AkuAuction	11539.5	This contract represents an auction with a time limit of 630 blocks (equivalent to 126 minutes). It was exploitable using our second attack, which resulted in a cost of 252 ETH with a maximum priority fee of 13 Gwei. The cost of the attack is much smaller than the contract's holdings.
0xd91ee91FD0f3fb15C9B9DD47F156396aa8C7c84B	AuctionHandler	599.0	This contract serves as an auction handler and can be vulnerable based on the `auctionEndTime` value and the bids set during an auction period. For it to be secure, one should ensure that `auctionEndTime` is so far in the future that applying our second attack would cost more than the bids in the auction. In practice, very short periods are used in this contract and it is vulnerable to our second attack.
0x432d26c295cE42f3999d8275b3107E34305Cd52A	O2LandSale	362.5	This land sale auction can be attacked by congesting merely 4 blocks. Although the auction period is short, it is limited to whitelisted users only, with no public auctions available. However, any of these whitelisted users can perform a congestion attack and both of our attack variants are economical in this case. Assuming the locks are 4 blocks, the gas hungry and verified single transaction attacks will cost 4.98 and 3.91 ETH respectively.
0x87acAE6dF21385A74ed4FB55A1a29354E9bdc6c1	VoyagerPass	158.8	Another auction contract with a short duration of 5400 seconds, which can be attacked by congesting less than 450 blocks. Attacking this auction would cost 182 ETH which is higher than the current balance of the contract, but one does not need to congest the entire duration of the auction. Even congesting the last quarter would make many participants unable to send their bids.
0x0193B85c38337EB90338Ed8660810ba66c548b62	AsterFi	59.2	This contract's vulnerability is due to its use of an unreliable source of randomness. The random seed is based on the block timestamp, block difficulty (which is a constant after the switch to proof-of-stake), and the **msg.sender** address. Thus, an attacker who can congest the blockchain is able to tamper with the randomness.
0x165f848F980309f6147b8adfC8589cc35c587Ca7	BitcoinCowsBridge	53.9	This contract relies on the block hash of the 14th block after the commitment call. Modifying the blockhash explicitly within the 14-block duration can impact the `shuffle()` random output. An attacker can use either of our two attacks to ensure this 14th block only contains a single function call by him, thus having his desired hash value. The attacks will cost 1.10 ETH which is executed by congesting one of the blocks to change the random output.
0x41d3d86a84c8507A7Bc14F2491ec4d188FA944E7	MoneyMakingOpportunity	45.7	This contract can be attacked by adding `n` new voters towards the end of the weekly auction, enabling the retrieval of double the invested money by congesting the network until the end of the week.

References

[1] G. Wood, “Ethereum: A secure decentalized generalised transaction ledger (berlin version),” https://ethereum.github.io/yellowpaper/paper.pdf, 2014.

[2] L. Heimbach and R. Wattenhofer, “Sok: Preventing transaction reordering manipulations in decentralized finance,” in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, ser. AFT ’22. ACM, Sep. 2022. [Online]. Available: http://dx.doi.org/10.1145/3558535.3559784

[3] P. McCorry, A. Hicks, and S. Meiklejohn, “Smart contracts for bribing miners,” Cryptology ePrint Archive, Paper 2018/581, 2018, https://eprint.iacr.org/2018/581. [Online]. Available: https://eprint.iacr.org/2018/581

[4] M. Tang and A. Zhang, “Transaction fee mining and mechanism design,” 2023.

[5] “Consensys: The inside story of the cryptokitties congestion crisis,” https://consensys.net/blog/news/the-inside-story-of-thecryptokitties-congestion-crisis/, accessed: 2023-08.

[6] E. Frangella, “Crypto black thursday: The good, the bad, and the ugly,” https://medium.com/aave/crypto-black-thursday-the-good-the-bad-andthe-ugly-7f2acebf2b83, accessed: 2023-08.

[7] “Eip-1559: Fee market change for eth 1.0 chain,” https:/ethereum-magicians.org/t/eip-1559-fee-market-change-for-eth-1-0-chain/2783, accessed: 2023-09.

[8] H. Chung and E. Shi, “Foundations of transaction fee mechanism design,” Cryptology ePrint Archive, Paper 2021/1474, 2021, https://eprint.iacr.org/2021/1474. [Online]. Available: https://eprint.iacr.org/2021/1474

[9] C. Sguanci and A. Sidiropoulos, “Mass exit attacks on the lightning network,” 2022.

[10] M. Khabbazian, T. Nadahalli, and R. Wattenhofer, “Timelocked bribing,” Cryptology ePrint Archive, Paper 2020/774, 2020, https://eprint.iacr.org/2020/774. [Online]. Available: https://eprint.iacr.org/2020/774

[11] J. Harris and A. Zohar, “Flood & loot: A systemic attack on the lightning network,” 2020.

[12] T. Roughgarden, “Transaction fee mechanism design for the ethereum blockchain: An economic analysis of eip-1559,” 2020.

[13] Etherscan, “Ethereum verified smart contracts,” 2023. [Online]. Available: https://etherscan.io/contractsVerified