Partitioning Ethereum without Eclipsing It

Hwanjo Heo
ETRI
Daejeon, South Korea
hwanjo@etri.re.kr

Seungwon Woo
ETRI/KAIST
Daejeon, South Korea
seungww@etri.re.kr

Taeung Yoon
KAIST
Daejeon, South Korea
yoontaeung@kaist.ac.kr

Min Suk Kang*
KAIST
Daejeon, South Korea
minsukk@kaist.ac.kr

Seungwon Shin
KAIST
Daejeon, South Korea
claude@kaist.ac.kr

***Abstract*—We present a practical partitioning attack, which we call Gethlighting, that isolates an Ethereum full node from the rest of the network for hours without having to occupy (or eclipse) all of the target's peer connections. In Gethlighting, an adversary controls only about a half (e.g., 25 out of total 50) of all peer connections of a target node, achieving powerful partitioning with a small attack budget of operating several inexpensive virtual machines. At the core of Gethlighting, its low-rate denial-of-service (DoS) strategy effectively stops the growth of local blockchain for hours while leaving other Ethereum node operations undisturbed. We analyze how subtle and insignificant delays incurred by a low-rate DoS can lead to a powerful blockchain partitioning attack. The practical impact of Gethlighting is discussed — i.e., the attack is scalable and low-cost (only about $5,714 for targeting all Ethereum full nodes concurrently for 24 hours), and extremely simple to launch. We demonstrate the feasibility of Gethlighting with full nodes in the Ethereum mainnet and testnet in both controlled and real-world experiments. We identify a number of fundamental system characteristics in Ethereum that enable Gethlighting attacks and propose countermeasures that require some protocol and client implementation enhancements. Ethereum Foundation has acknowledged this vulnerability in September 2022 and one of our countermeasures has been accepted as a hotfix for Geth 1.11.0.**

I. Introduction

Highly reliable network connectivity across distributed nodes is a critical system property for blockchain implementations that must persist even in the face of catastrophic network failures or attacks. Thus, public, permissionless blockchains are designed and operated with the goal of achieving reliable peer-to-peer connectivity. For example, a Bitcoin node by default establishes up to 125 peer connections to achieve its network reliability, while an Ethereum node typically makes up to 50 connections.

The security community nevertheless continuously discovers new attack vectors that *partition* one or more peer nodes in these densely connected blockchain networks, demonstrating that network-layer attacks on blockchains can result in the violation of safety properties [3], [22], [33], [37]. At a high level, these partitioning attacks need an *airtight* control of a target node's peer connections (a.k.a. *eclipsing*). With complete control over the consensus information to a target node, an adversary can delay (either temporarily or indefinitely) the blockchain's canonical chain information and also feed any arbitrary transactions/blocks tailored for consensus violations, such as double spending attacks. For example, the Bitcoin eclipse attack [22] demonstrates that a botnet master with a small size (e.g., roughly 4.6K) botnet can easily control all peer connections to a target node. The Bitcoin hijacking [3] and Erebus [37] attacks demonstrate that a network adversary (e.g., a malicious ISP) can also take complete control over a target's peer connections. SyncAttack [33] shows that even partitioning an entire blockchain network is possible.

We introduce an Ethereum partitioning attack, dubbed Gethlighting, that, unlike these previous partitioning attacks, does *not* require an adversary to have complete control over all peer connections of a target node. Our attack only requires to control approximately half of a target Ethereum node's 50 peer connections, much less than what previous attacks demand, to successfully demonstrate a practical partitioning attack in the Ethereum mainnet. This essentially means that the other half of peer connections remain freely available for block delivery while the target node is partitioned by Gethlighting.

Controlling only about half of a target node's peer connections, a Gethlighting adversary causes an effective *low-rate denial-of-service (DoS)* disruption to significantly slow down the growth of the blockchain at the target for up to a couple of hours. We present a highly surgical DoS attack that disturbs the chain growth of the target but leaves other operations undisrupted. The target still receives every message (including new blocks and transactions) from its other benign peers without any major disruption (unlike a recent Ethereum DoS attack [27]). Our attack gently disturbs the target's multi-peer message handling logic so that it fails to process a new block within a pre-defined timeout *once in a while* but not frequently. Such a rare timed-out reception of a new block, however, is sufficient to stop the growth of the local blockchain since the block-level state transition is not commutative in the blockchain model [41]. Our in-depth study explains why Gethlighting's low-rate DoS strategy of utilizing invalid transactions effectively partitions a target node while many other types of low-cost DoS attacks are futile.

Figure 1 shows an example of what happens to a real target Ethereum node in the mainnet during a Gethlighting attack that controls 32 of the target's 50 connections. The primary effect of this attack shown in this example is that the target's chain stops growing for 521 blocks (about 1 hour 22 min) in the worst case; see that one new block is generated at around $T = 2,800$ sec, learned by the target instantly, but inserted to the target's chain at around $T = 7,800$ sec! Note that the

*Corresponding author

Network and Distributed System Security (NDSS) Symposium 2023
27 February - 3 March 2023, San Diego, CA, USA
ISBN 1-891562-83-5
https://dx.doi.org/10.14722/ndss.2023.24465
www.ndss-symposium.org

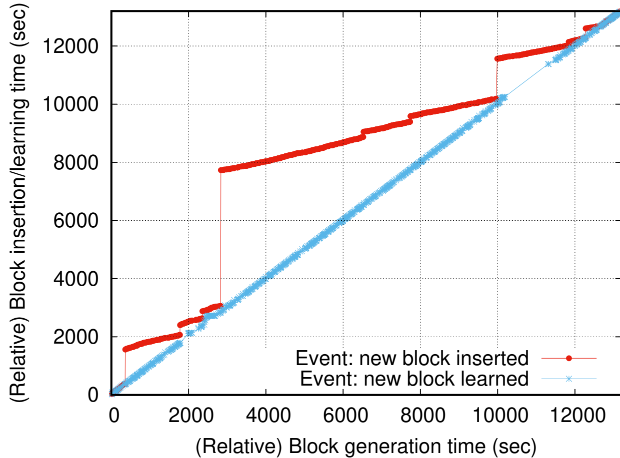


Fig. 1: An example snapshot of two types of events experienced by a mainnet target node during a Gethlighting attack: learning a new block (blue stars) and inserting the new block into its chain (red dots). The relative time of these events is presented on the Y-axis while the relative time of their initial block generation is on the X-axis. Gethlighting barely delays block learning (i.e., most blue stars are on the unit-slope straight line) but significantly delays block insertion up to 4,842 sec (1h22m), implying that the target is seriously partitioned.

delays experienced for learning new blocks rarely exceeds 10 seconds even when the target is under attack as it still receives timely block information from its 18 other benign peers.

The main contribution of this paper can be summarized as follows:

1) We introduce the Ethereum Gethlighting attack (§III), a new partitioning attack that does not require complete control of a target node's peer connections. This relaxation lowers the bar for partitioning attacks significantly. This is the first partitioning attack that does not require a total eclipse of peer connections since an earlier attack by Wüst and Gervais [42] in 2016 — i.e., a powerful attack that was once effective but has been prevented after a quick hotfix [35]. In Section V, we demonstrate the feasibility of the Gethlighting attack through extensive experiments in the Ethereum testnet and mainnet. We launch and confirm full-scale Gethlighting attacks against some critical nodes in the testnet and our own nodes in the mainnet. We also carefully select a few critical nodes actively operated in the mainnet and run a basic attack feasibility test without causing any damage to them. Since Gethlighting never eclipses any nodes, it renders existing eclipse mitigation techniques, such as manually configuring static peer connections to reliable peers or networks (e.g., routing-aware peering [38], trusted static peer connections [23], a hijacking-resistant relay SABRE [2], or traditional relay networks [16], [18]) completely ineffective.
2) We conduct an in-depth examination of the low-rate denial-of-service strategy of the Gethlighting attack. In particular, we investigate the per-peer isolation design in the up-to-date Ethereum clients and identify a subtle scheduling condition that fails a block insertion, which is exploited in the Gethlighting attack. In addition, we identify and analyze four more characteristics of the current Ethereum client implementation that contribute to the effective Gethlighting attack (§IV).
3) We assess the proposed attack's high-risk impact on the existing Ethereum network. First, Gethlighting is a low-cost attack. Since not having to eclipse a target greatly reduces attack costs, it can be carried out by anyone with several inexpensive virtual machines. Second, Gethlighting can simultaneously target multiple Ethereum nodes using the same set of virtual machines, making the attack easily scalable. Our simple calculation estimates that it would require only about $5,714 for targeting all Ethereum full nodes concurrently for a full day! Third, Gethlighting is simple to prepare and execute (e.g., no knowledge of network topology or routing information is required), making it an attractive attack vector for unskilled adversaries (§VI). Last, if an adversary has some mining capacity, she can easily feed her own blocks to the target and establish a temporary fork (§V-B), making Gethlighting a good launching pad for attacks like double spending [33] or stubborn mining [31].
4) We propose a number of countermeasures that would be necessary to mitigate the Gethlighting attack (§VII). Since the five characteristics of Ethereum we analyze are fundamental system features, not bugs, no simple, complete solution appears to be available. We instead present several minor tweaks to the Ethereum protocol and client implementation parameters to reduce the effectiveness of the attack. Our analysis suggests that all of these tweaks come with some caveats; e.g., potentially opening up new attack surfaces. Thus, careful, large-scale testing of these potential countermeasures must be followed up to determine their optimal combination for the Ethereum network.

Ethereum Foundation has acknowledged the Gethlighting vulnerability in September 2022. In response to our suggestion in Section VII-B (see the corresponding pull request [34]), the Geth team has included a hotfix for Geth 1.11.0 [20].

For the demonstration of Gethlighting in this paper, we mainly focus on the Go Ethereum client implementation (commonly known as Geth), as this implementation is used by the vast majority (80.7%–91.9%[1] during the first half of 2022) of Ethereum nodes. Because Gethlighting does not entirely rely on the Geth's specific implementation flaws but also on the Ethereum protocol design tradeoffs, other Ethereum client implementations could be vulnerable to Gethlighting as well. We conduct a small proof-of-concept experiment on OpenEthereum, the second most popular Ethereum client (6.5% of Ethereum nodes), and demonstrate the attack's feasibility in Appendix A.

II. Background

In this section, we provide some necessary context for understanding our Gethlighting attack: Ethereum's peer-to-peer network structure (§II-A), peer message handling logic (§II-B), and gossiping strategies (§II-C).

A. Ethereum Peer-to-peer Network

Ethereum mainnet consists of thousands of fully-validating nodes. By default, each network node makes up to 50 peer

[1] https://etherscan.io/nodetracker

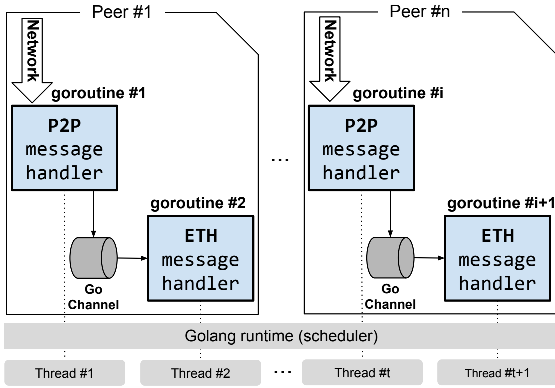


Fig. 2: Peer message handling goroutines of Geth.

connections, one-third of which are outgoing and the other two-thirds are incoming. An Ethereum node selects its outgoing peers at random from a list of known addresses provided by the Ethereum's discovery protocol [8]. If a node has already established the maximum number of incoming peer connections, all subsequent inbound connection requests are rejected. Notice that this policy is different from that of Bitcoin, which unconditionally evicts a randomly chosen existing peer connection to accept a new incoming connection [6].

B. Peer Message Handling Logic

An Ethereum node communicates with multiple peer nodes concurrently. Since some peer nodes may have poor performance, or demonstrate selfish behaviors, implementing an efficient and robust peer message handling is critical.

Geth employs an efficient concurrency model that utilizes the lightweight threads (i.e., goroutines) in Go language to separate peer connections from each other. Figure 2 shows that each peer connection is assigned two goroutines: (1) a P2P message handler that receives messages from each peer and (2) an ETH message handler that processes the enclosed Ethereum subprotocol messages, if exist.[2] An unbuffered Go channel delivers subprotocol messages between the two message handling goroutines in a blocking manner.

C. Gossiping Messages: Unsolicited vs. Solicited

Ethereum broadcasts all valid blocks and transactions through the entire network via a gossip protocol. When gossiping about a newly received block or transaction, an Ethereum Geth client uses two different mechanisms: unsolicited or solicited. A Geth node forwards the full block/transaction content only to a small subset of its neighboring peers, and we call it *unsolicited* block/transaction propagation. The current Geth implementation chooses a subset of size 7 ($= \sqrt{50-1}$), which is decided in a heuristic manner [25]. For the rest of 42 neighboring peers, the Geth node sends a *solicited* message, which contains only the hash of the new block/transaction. The receiving peer of the solicited message should actively fetch the new blocks/transactions from the sender.

III. The Gethlighting Attack

In this section, we introduce our Gethlighting attack. We first present the threat model we consider in this paper and explain the four steps of the Gethlighting attack (depicted in Figure 3).

Threat model. The attack goal of Gethlighting is to disrupt the chain growth at a targeted Ethereum node for significant amount of time (up to a few hours). The target node is a full node that accepts connections from other Ethereum nodes with a public IP address. Gethlighting makes only about a half of the peer connections of a target node and this requires some inexpensive virtual machines (VMs) in public clouds.

Creating a temporary partition via Gethlighting does *not* require any mining power at all. Yet, if a Gethlighting adversary wishes to create a fork with her own blocks through the last optional Step-④, she should have some mining power to generate her-own blocks. With some significant mining capacity, an adversary may use Gethlighting as a stepping stone for other attacks (e.g., double spending [33], stubborn mining [31]), which are out of scope of this paper.

A. Step-①: Making Some Connections to the Target

The Gethlighting adversary uses her VMs as Ethereum nodes and patiently waits to occupy some of the target's peer connections. Our attack VMs run a slightly modified Geth client to do so. The target Ethereum client may reject our peer connection requests when its inbound connections are already completely filled with 34 existing inbound peer connections. When an existing benign peer connection is disconnected from the target node, our attack VMs have a chance to make a connection to the target. Thus, the attack VMs may need to retry to make enough connections to the target. Since Gethlighting does not require to fully occupy all peer connections, the adversary can choose to stop making more connections to the target when the number of peer connections to the target is considered enough; we show that a half of the maximum number of connections (i.e., 25) is a good starting point. We observe that it takes at most a day to occupy at least 25 peer connections when targeting critical Ethereum nodes in the Rinkeby Ethereum testnet (see Section V-C), a little longer in the mainnet (see Appendix E).

B. Step-②: Causing Mild Disruption to Time out a Block

In this step, the attack aims to cause some mild disruption to the target's block propagation so that a single block is timed out at the target node. The target node may miss a few more blocks due to the attack but one missing block is sufficient for the attack. Only a single missing block sufficiently stops the growth of the local blockchain since the block-level state transition is not commutative in the blockchain model [41]. Thus, a state replication process at the target should wait for the single timed-out block before attaching any subsequent blocks.

The disruption should be mild enough so that most of block propagation (and other Ethereum operations) is unaffected.

[2] An Ethereum P2P message (i.e., DEVp2p) may contain a subprotocol message such as ETH (Ethereum wire protocol [12]) or LES (Light Ethereum subprotocol [10]).

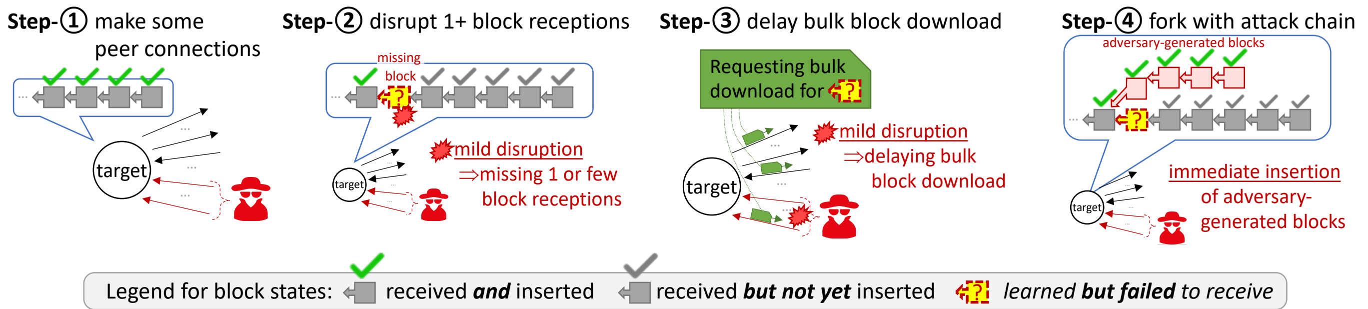


Fig. 3: The Gethlighting attack overview.

We present a new low-rate denial-of-service (DoS) strategy to achieve this. A *TX-flooding* strategy makes successive submission of Ethereum transactions, hoping to cause a low-rate DoS problem at the target. As the rate of flooding is low, the target node still behaves normally (e.g., receiving and processing most of Ethereum messages without noticeable delays); however, it fails to process some blocks it receives from time to time. When the target is under our TX-flooding attack, it sometimes causes enough delay (e.g., 5 seconds in the Geth's default setting) for processing a newly received block so that the block is timed-out eventually. We show that it is, in fact, easy to induce such a mild DoS in the target's message processing routine by sending a batch of Ethereum transactions that are all eventually invalidated by the target node. Due to the Ethereum clients' resource management scheme (see Section IV-A), the target sometimes misses a block.

C. Step-③: Delaying Bulk Block Downloads

Once a single block reception has timed out at the target node, its chain growth stops because the missing block prevents all subsequently received blocks from being chained back to the genesis block. To retrieve the missing block, the target now begins the bulk block download process[3] as the last resort of retrieving the missing block. The bulk download mechanism in Ethereum is designed for the recovery of major delivery failures or initial block downloads. Thus, it is optimized for downloading a long sequence of blocks and catching up the synchronization with other peers.

The TX-flooding strategy is used in Step-③ as well to keep delivering some workload and mildly burden the target node. This renders the retrieval requests to fail even when asked to a benign peer. In particular, the bulk block download routine can be executed multiple times, making it hard to get out of the recovery mode of the target node. We show that the block delivery can be, therefore, delayed for up to a few hours with this strategy.

D. Step-④: Forking with an Attacker-Mined Chain

While the target node is partitioned from the rest of the Ethereum network, our Gethlighting adversary can also reliably feed the full copies of her own mined blocks to the target. The adversary-generated blocks are sent to the target as whole blocks (i.e., unsolicited block delivery) to ensure that they are accepted by the target without experiencing any delays. As a result, the adversary's chain becomes the longest branch from the target's perspective as long as the missing blocks are not yet downloaded.

[3]The official name of this process is *block synchronization*. In this paper, we call it a bulk block download to better highlight its download design and differentiate it from the regular block propagation process.

TABLE I: Five key characteristics of Ethereum clients that enable the Gethlighting attack.

Ethereum's Characteristics	Step-①	Step-②	Step-③
[EC1] Pretty-Strong-yet-Limited Per-Peer Isolation		✓	✓
[EC2] Optimized for Fast and Reliable Block Propagation		✓	
[EC3] Gracious Handling of Invalid Transactions		✓	✓
[EC4] Limited Buffer Size for Out-of-order Blocks		✓	
[EC5] Block Propagation Only via Solicited Messages	✓		

IV. Technical Underpinnings

In this section, we discuss five technical characteristics (which we label as **[EC]**—Ethereum Characteristics) of the Ethereum client design and implementation that enable our Gethlighting attack. Each of them is discussed in detail in each subsection. We use the term "characteristics" instead of "vulnerabilities" because they are carefully designed, critical system features for addressing the Ethereum network's efficiency, robustness, and security challenges.

Table I summarizes how each Ethereum's characteristic contributes to the corresponding attack steps.

A. [EC1] *Pretty-Strong-yet-Limited Per-Peer Isolation*

A blockchain node communicates concurrently with multiple peer nodes. One ideal property for multi-peer scheduling in blockchains would be a *strong per-peer isolation* between peers at a local node. Without proper isolation between peers, malicious peers may cause an adverse effect on the delivery and processing of messages (e.g., new block information) from other benign peers.

Strong per-peer *fairness*. Our thorough analysis of Geth (version 1.10.20) shows that its concurrent message handling logic for serving multiple peers achieves, in fact, *strong per-peer fairness*. That is, each user connection is guaranteed to have its fair share bandwidth of the resources at the target node. Several careful design choices found in the Geth's message handling logic make it fair across peers:

(i) *No shared queues between peers.* All network protocol and application buffers are separately provided for each peer

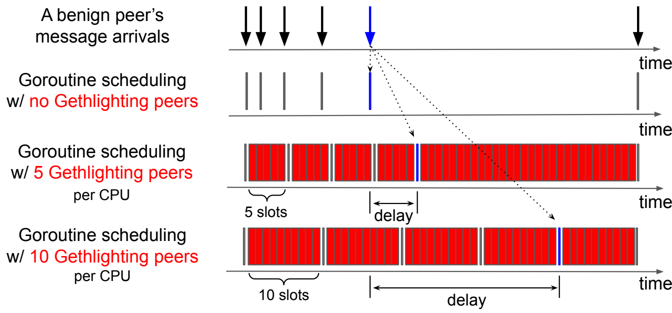


Fig. 4: A simplified illustration of goroutine scheduling in one CPU core during the Gethlighting attack.

connection. Ethereum messages from each remote peer are handled by two dedicated goroutines, providing a separate message buffer for each (see Figure 2).

(ii) *Round-robin scheduling.* The message handling goroutines are scheduled by the Go runtime scheduler in a round-robin fashion as each CPU has its own FIFO local run queue (LRQ) from which goroutines in ready are dispatched for execution.

(iii) *Preemptive scheduling.* The Go runtime scheduler implements a preemptive scheduling. That is, a message handling goroutine is preempted by the blocking I/O (i.e., waiting for the next message to arrive) or by reaching its typical time-quantum of 20 msec.[4]

This strong per-peer fairness in Geth makes it highly robust to a wide range of denial-of-service (DoS) attacks; see our exhaustive experiments in Section V-A, which shows that flooding with various Ethereum message types are all ineffective, except the proposed TX-flooding attack. Per-peer fairness in Geth provides pretty strong isolation because it offers the guaranteed message handling performance per peer that is lower-bounded by the *total number of messages* sent to a target Geth client divided by the number of all peers (e.g., maximum 50 in the default setting).[5] This means that an adversary's effect on the target is bounded by the number of peer connections she controls.

When per-peer fairness isn't enough. Our TX-flooding attack demonstration, however, shows that the current Geth's per-peer fairness policy and the default parameters for peers and their messages still allow a certain type of denial-of-service attacks. Let us get into some more details and understand why per-peer fairness is not strong enough to address the TX-flooding attack, unfortunately.

Incoming peer messages accumulate waiting time in the queue if the arrival rate λ (i.e., the rate at which messages are sent from the remote node) exceeds the service rate μ (i.e., the rate at which messages are processed at the local node). In normal cases, μ (per a normal peer connection) depends on the local node's system performance and the amount of work that should be done for those messages. The problem is that the service rate μ is bounded to $1/(k \times s)$ with TX-flooding by a Gethlighting adversary, where k is the number of Gethlighting peers divided by the number of CPU cores of the target and s denotes a system parameter for the maximum time bound for each scheduling; 20 msec measured in Geth. Figure 4 illustrates a simplified example of this: The fifth message (blue arrow) from the benign peer is handled with a significant delay in the presence of 10 Gethlighting peers (per CPU) because each Gethlighting peer occupies the entire time slot (i.e., 20 msec) of its fair share of scheduling, rendering 200 msec (i.e., 10 $\times$ 20 msec) serving as the minimum inter-service time of benign peer messages. It should be noted that attack peers cannot *indefinitely* delay benign peer's messages or take more scheduling opportunities than their fair share.

The bounded service rate induced by a Gethlighting adversary renders delayed message delivery when messages arrive more frequently (than the bound). A local node's fetch timer, which is set up for a specific request to a remote peer, expires because the response message experiences a long queuing delay (as depicted in Figure 4) inside the local node.

From our back-of-the-envelop calculation of the required number of attack peers for TX-flooding, we learn that with 20 attack peers against a target node with 4 CPUs (thus, $k = 20/4 = 5$) any benign peer with message rates higher than 10 messages/s would experience delays in their message handling at the target. This result is well aligned with our main evaluation of the TX-flooding strategy discussed in Section V-A.

B. **[EC2]** *Optimized for Fast and Reliable Block Propagation*

Fast and reliable block propagation — i.e., the process of disseminating newly mined blocks to the entire network nodes — is crucial for blockchains. This is particularly true for Ethereum as its blocktime (12–14 seconds) is more than an order of magnitude shorter than that of its main competitors (e.g., 10 minutes in Bitcoin). Thus, the Ethereum protocol and client implementations have continuously evolved over time to optimize for fast block propagation, and the current Ethereum blocks propagate to 90% of peer nodes in less than 1.5 seconds [15].

Let us summarize two layers of optimizations for fast and reliable block propagation in Ethereum:

1) **Asking *only one* peer for a full block.** A Geth client batch-processes multiple messages for the same solicited block delivery that have been received within a 0.4-second time window.[6] When asking for a full block information, a client node sends a request only to one peer from each batch. This removes any duplicate download of full block information from two or more peers, thus minimizing the network bandwidth for block propagation.
2) ***Limiting retrials* for a new block.** When a request for a solicited block is unanswered within a timeout (e.g., 5 seconds[6] in the current Geth version 1.10.20), it is considered failed and the *entire batch* of solicited announcements are discarded. The requesting peer may retry downloading the same block with another batch of solicited announcements that came after the first batch; however, the retries

[4]A goroutine that runs for 10 msec is labeled preemptible, and the actual scheduling-out process takes place later, once a few more conditions are satisfied. We observe that the actual time slot durations are very close to an average of 20 msec via our execution tracing in Appendix D.

[5]Because the message handling goroutines are scheduled in a round-robin fashion, at least $1/n$ of the time is guaranteed to be spent serving one specific peer's messages, where n is the number of peers.

[6] See Appendix F for the hard-coded timeout values in Geth. It is also discussed why simply increasing the timeout value is not an effective countermeasure against Gethlighting.

are practically limited only to a couple times [7] because the Ethereum network is fast enough to finish a block propagation in 1-2 seconds. This particular policy has been deployed to protect peers from a form of denial-of-service attacks [9]. That is, by limiting the number of retrials, a peer node can ignore most of the maliciously repeated solicited announcements for the same hash of a non-existent block.

Note that these optimizations have been evolved over time and work nicely for most cases. We argue that these optimizations, however, unfortunately make Ethereum clients vulnerable to our Gethlighting attack by helping a Gethlighting adversary easily make Ethereum clients miss a single block reception.

As a result of these optimizations, an Ethereum peer node has *only a couple of chances* of receiving a full block in its initial propagation phase (i.e., a few seconds after its creation), as we empirically confirm in Appendix B. Missing this scant opportunity, a node should later rely on the bulk download of blocks, which involves much more expensive interactions with its peers.

Let us provide some more details. When the existence of a new block is informed, `block_fetcher` in Geth collects all the announcements with the same block hash from different peers for 0.4 seconds. After gathering the batch, one request for the block header is sent to a single peer that is selected at random. If the request is not answered within the 5-second timeout window, *all* the announcements in the batch are discarded together. Ethereum network is known to be highly efficient in block propagation (e.g., more than 80% of nodes receive a new block information within 1-second window since the block creation [15]) and thus a peer node typically collects only a couple of batches of solicited announcements after all!

C. **[EC3]** *Gracious Handling of Invalid Transactions*

In blockchains, when transactions are deemed invalid, they ought to be ignored and dropped by peers as early as possible in a system. A transaction is said to be syntactically invalid when it has missing fields, wrong signatures, or incorrect proof-of-work. In contrast, a semantically invalid transaction has an instruction (e.g., transferring ETH from one account to another) that is considered unacceptable according to the canonical chain and other transactions stored in a node. For example, a transaction that spends some ETH from a zero-balance account is considered semantically invalid. The Gethlighting attack sends many such *syntactically-correct-yet-semantically-invalid* transactions to a target node for TX-flooding.

We find that this is possible due to the policy of *gracious handling* of invalid transactions, which is a distinctive policy found in Ethereum. A not-so-gracious handling of invalid transactions (found in Bitcoin) would penalize the peers that send such invalid (despite their syntactic correctness) transactions. A strict handling of invalid transactions could even disconnect a peer when it sends any invalid transactions. Ethereum, however, explicitly refuses to employ any penalty for sending such invalid transactions but implements the gracious handling policy. That is, as long as a transaction is syntactically correct, it never penalizes peers for sending invalid transactions. Instead, Ethereum decides to handle all (both valid and invalid) transactions as efficiently as possible so that peer nodes do not experience disruptions due to invalid transactions.

There exists, in fact, a good reason for this Ethereum's generosity towards invalid transactions. In an account-based model, it is often uncertain whether an invalid transaction is sent with a malicious *intent* or not. One clearly invalid transaction in a node may be considered valid in another node with a different blockchain state. As temporary forks exist in the Ethereum blockchain network, one cannot easily conclude that the sender (or forwarder) of an invalid transaction has a malicious intent behind it. For example, an Ethereum node, which is synchronized up to the block height h, may propagate a transaction t that is believed to be valid. However, another node that is one block behind (i.e., synchronized up to $h - 1$) may conclude that t is *invalid*. As seen in the example, it is inappropriate to hastily blame and penalize a peer for sending invalid-looking transactions because it can be a perfectly honest, protocol-abiding peer that believes the validity of the transactions. Ethereum chooses to handle such transactions graciously and completely avoid penalizing potentially benign peers. Our Gethlighting attack exploits this very policy decision.

D. **[EC4]** *Limited Buffer Size for Out-of-order Blocks*

Our Gethlighting attack in Step-② actively triggers a target node to miss only one or few blocks while *allowing* most or all subsequent block propagation. This causes blocks arrive *out-of-order* at a target node and then all received blocks after the first missing block are sent to a local buffer, waiting for the missing block to be downloaded. Buffering out-of-order blocks saves time for re-downloading out-of-order blocks because all the buffered blocks can be immediately inserted to the blockchain as soon as the missing block is downloaded. The larger the buffer size for out-of-order blocks, the faster a target node can recover from our Gethlighting attack.

However, the buffer size should not be set too large due to the risk of denial-of-service attacks; e.g., a perpetrator may overflow the buffer capacity with many out-of-order blocks. The current Geth implementation stores up to 32 out-of-order blocks. Due to this limited buffer, when the Gethlighting attack successfully delays the download of a single block more than 6.5 minutes (i.e., about 32 block times), out-of-order blocks start to drop at a target node. This means that some out-of-order blocks are not ready in the target node at the time when a missing block is finally downloaded and they need to be downloaded again.

***Back-to-back* bulk block downloads.** This small buffer size can potentially put the target node into a series of back-to-back bulk block downloads (as in Step-②), further extending the partitioning attack. At first, one missing block due to Gethlighting causes to drop some out-of-order blocks. Then, when the missing block is received, a target node often re-enters the bulk block download phase because it has dropped some of the more recent out-of-order blocks. This can repeat multiple times, rendering the target node to experience multiple rounds of delayed bulk block downloads.

[7] It is determined by how widely distributed the new block announcement arrivals (from neighboring peers) are. See Appendix B for the empirical distribution.

E. **[EC5]** *Block Propagation Only via Solicited Messages*

The current Geth node delivers a newly received block to only a small number of neighboring peers *unsolicitedly* (see its definition in Section II-C). The Gethlighting attack works when a target node receives a new block *exclusively* via solicited messages. This is when the target receives only the hash of the new block (via solicited messages) from all its peers but no full-block information (via unsolicited messages). Such solicited-only block propagation happens quite frequently to any node in Ethereum and thus our Gethlighting can opportunistically attack any target node without waiting too much.

The probability that a target node receives solicited block messages from all n peers can be modeled as $\Pi_{i=1}^{n}\left(1-\frac{1}{\sqrt{m_i-1}}\right)$, where m_i denotes the number of peer connections of the target's peer i among all n peers with some simplifying assumptions (e.g., all peers are always ready to relay up-to-date blocks). By default, a Geth node makes peering with up to 50 other Ethereum nodes (i.e., $n = m_i = 50$ for all i); however, as a Gethlighting adversary controls one or more peers of a target Geth node and does not send unsolicited messages through them, the effective n for the probability decreases to the number of benign peers of the target. For example, when a target has 25 attack peer connections ($n = 25$), the probability that a new block is received exclusively via solicited messages becomes 2.12% and it increases to 8.48% when the target has 34 attack peers ($n = 16$). Although the probability for a single block still seems low, a Gethlighting adversary can wait for several new blocks until the target node eventually receives a block exclusively via solicited messages. The expected waiting time to the first solicited-only block propagation at the target is 8.67 minutes (about 47 blocks) for a target with 25 attack peers and 2.36 minutes (about 12 blocks) for a target with 34 attack peers.

In theory, if a target Geth node is modified to have more than the default 50 peer connections [11], n increases accordingly and the above probability diminishes, which would increase the time till the Gethlighting attack triggers Step-②. However, in practice, Gethlighting is still effective against target nodes with more than 50 peer nodes because not all peers are always ready to forward new blocks in the real Ethereum networks; see our empirical evaluation in the mainnet (Section V-A) and testnet (Section V-C).

V. Mainnet and Testnet Experiments

This section lists our extensive experiment results of the Gethlighting attacks. We first conduct safe, isolated experiments with our own Ethereum *mainnet* client nodes for in-depth analysis of the proposed attack (§V-A). Extending this attack, we attempt to inject adversary-mined blocks to our own target nodes while Gethlighting them (§V-B). We then conduct our full-scale Gethlighting attacks against several important nodes in the Ethereum testnets and confirm its real-world impact (§V-C). Finally, we test the feasibility of Gethlighting against the real-world important nodes in the Ethereum mainnet (§V-D).

In all our evaluations, we carefully design and execute these experiments to minimize potential negative impact (if any) on the Ethereum mainnet (§V-E).

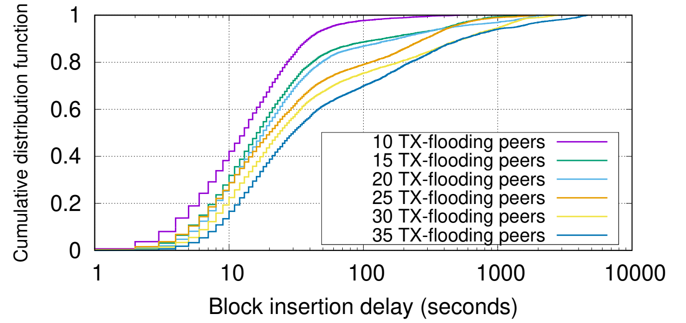


Fig. 5: Block insertion delays experienced by a target node under TX-flooding attacks.

Common attack setup. We implement Gethlighting attack strategies by adding approximately 200 source lines of code (SLOC) to the official Ethereum implementation, Geth v1.10.20. The essential attack logic is embedded in only 20 SLOC. For TX-flooding attacks, we generate a transaction journal file, comprising 96,000 invalid transactions transferring 1 ETH from a zero-balance account to another account, in advance and input during Geth's bootstrapping.

Four `t2.large` instances, having 2 × Intel CPUs, 8 GB memory, and 100 GB general purpose HDD storage each, are provisioned to accommodate up to 40 attack nodes (i.e., 10 adversarial Geth processes per each VM). Gethlighting client nodes utilize only little resource footprint since they can stay nearly dormant, with the exception of 15-second periodic ping-pong, without worrying about being disconnected from the attack target. See Appendix G for more discussion on this.

A. Controlled Mainnet Experiments and In-depth Analysis

We first test the Gethlighting attacks in a safe, controlled setup where all the targets are under our control. To make our attack targets have realistic internal states, we let our attack target nodes connect to the Ethereum mainnet and receive transaction and block information from it (but not sending any to the mainnet). Our target node is hosted by an Amazon EC2 `i3.xlarge` instance, which satisfies the *recommended* hardware specifications from the official Ethereum foundation[8]; i.e., 4 × Intel Xeon CPUs @2.3Ghz, 30.5 GB memory, and 1 TB NVMe SSD storage.

We launch six independently executed experiments of Gethlighting with the different numbers of malicious peers. Each experiment is performed for 24 hours after synchronizing recent blocks in the target node and connecting all adversarial nodes to the target.

Figure 5 shows the cumulative distribution of block insertion delays at the target node. With 10 attacker's connections, 97% of blocks are inserted in less than 100 seconds since their creation. This decreases to 85% when Gethlighting makes 15 connections. Also, we observe the maximum delay of 1,400 seconds with 15 Gethlighting peers. When we utilize 35 peers for TX-flooding, 31% of blocks experience longer than 100-second delay, and the maximum we observe reaches 4,663 seconds. To sum, we empirically demonstrate that as more attack nodes are used for TX-flooding attacks, a longer partitioning duration is expected.

[8]https://ethereum.org/en/developers/docs/nodes-and-clients/

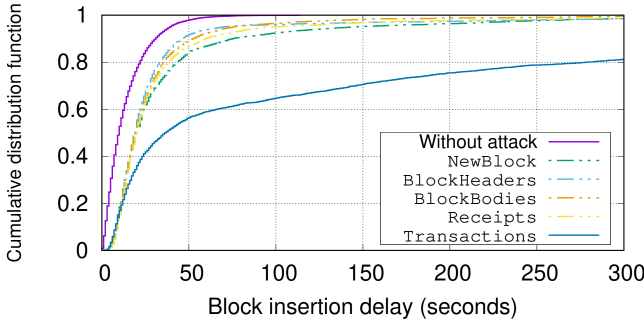


Fig. 6: Block insertion delays experienced by a target under low-rate DoS attacks with various Ethereum messages. Our TX-flooding strategy (i.e., low-rate DoS with transactions) clearly stands out. We show only the top five most effective Ethereum messages; all others make worse attacks.

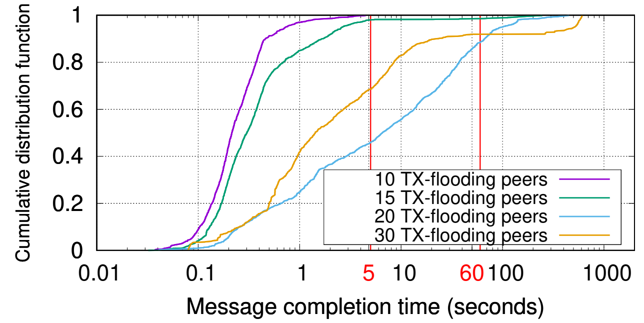


Fig. 7: Completion time for benign peer messages. Two timeout values for Step-② and Step-③ are indicated by two red vertical lines.

How is TX-flooding different from *naïve DoS* attacks? Our choice of Ethereum *transactions* (hence called 'TX'-flooding) among many other Ethereum message types (e.g., informing a block, requesting or responding block parts) for our attack strategy may seem like an arbitrary decision; however, it is our deliberate, well thought-out design choice. Here, we empirically show this.

We first implement many different types of DoS attacks with different Ethereum message types. To be specific, we execute simple back-to-back message flooding attacks with a comprehensive set of the up-to-date Ethereum message types (i.e., ETH/67[9]). A set of attack payloads comprising all 13 message types are pre-generated synthetically to meet the message size limit of 10 MB for a fair comparison with our TX-flooding attack. Message types `Status` and `GetBlockHeaders` are the only exceptions because the Ethereum protocol does not allow us to fill up a 10 MB message with these message types.

Figure 6 shows the cumulative distribution of block insertion delays at the target node when under various DoS attacks we test. Our TX-flooding is evidently far more effective than all other DoS attacks with different message types, making it the only effective flooding attack vector for our Gethlighting attacks. Note again that all these other flooding attacks also send the same size (i.e., 10 MB) messages to the target but they cause only a marginal impact on block insertion. See Appendix C for details.

Why the more attack connections, the longer partitioning? To find out what really causes the trend in Figure 5 (i.e., the more attack peers, the more powerful attacks), we conduct additional controlled experiments. For controlled experiments, unlike the ones in Figure 5, we directly manage both the benign and malicious peers of a target. We set the number of benign peer connections to 5 while we run 8 independent executions of TX-flooding attacks with 5 to 40 (with a step of 5) Gethlighting peers for a 1-hour period.

Figure 7 depicts the distribution of Ethereum message completion times for *benign* peers during each controlled attack execution. The message completion time quantifies the amount of time it takes from start to finish (i.e., from the time the message is sent from a remote node until it is completely processed by the target node). We find that when there are more TX-flooding peers, it takes longer to process an Ethereum message (e.g., informing new blocks and transactions). As a result, some of those messages are not delivered before the expiration of our attack's two critical timers (Section III): A *5-second timer* expiration (see a red vertical line at 5 sec in Figure 7) in retrieving single block information is an important triggering event in accomplishing Step-②. If a request for bulk block information is not responded to within the estimated round-trip time (e.g., one minute[10]), the retrieval request fails in Step-③ (see another red vertical line at 60 sec in Figure 7). The failed request is then forwarded to other peers, increasing the total partitioning duration. Our detailed analysis of why such delay occurs for our TX-flooding is found in Appendix D.

Figure 7 also supports our claim of mild disruption via TX-flooding. As shown in the figure, Gethlighting does *not* interfere with the processing of benign messages to the extent that the majority of benign messages miss the target's processing deadlines. Only a small fraction of benign messages violate the timeout; for example, approximately 10% or less of benign messages violate the 60-second timeout in all attack configurations and only about 2% violate the 5-second timeout when attacking with 15 peers.

Attacking a target modified to have more than 50 peers? A modified target Geth node can make more than 50 (i.e., default `maxpeers` value) peer connections. First of all, having more peer connections at a target does *not* reduce the Gethlighting attack strength because the delay in handling messages only depends on the number of attack peers, not benign ones. However, the overall attack effectiveness can be diminished because it becomes *harder* to trigger the condition for Step-② in Gethlighting when the target has more benign peers that can send unsolicited blocks (see Section IV-E).

To measure the reduced attack effectiveness with targets with more peers, we launch three independently executed experiments of Gethlighting by configuring the target Geth node's `maxpeers` to 50 (default), 100, and 200. The cumulative distribution of block insertion delays (collected for 24 hours) in Figure 8 shows that the Gethlighting attack is, by and large, highly effective against all target clients with

[9]https://github.com/ethereum/devp2p/blob/master/caps/eth.md

[10]We observe that the value is set close to one minute regardless of the actual network round-trip time, which is far smaller.

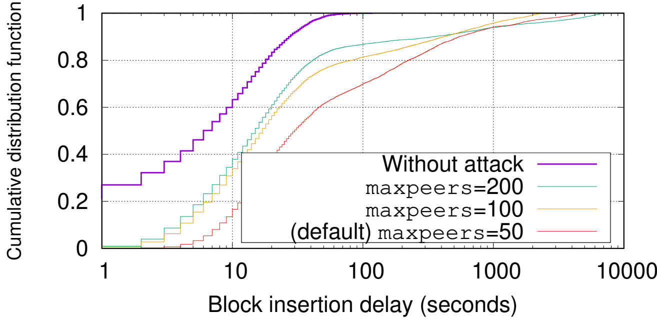


Fig. 8: Block insertion delays under TX-flooding attacks with configuring `maxpeers` to 50, 100, and 200 at the target.

different `maxpeers`. The overall delay is somewhat decreased with more numbers of peer connections as the median block insertion delays are 85, 17, and 15 seconds for 50, 100, and 200 `maxpeers`, respectively. However, the maximum block insertion delays (i.e., the worst-case partitioning duration) are 4,663, 2,502, and 6,892 seconds for 50, 100, and 200 `maxpeers`, respectively, showing the effectiveness of partitioning attacks across all `maxpeers` setups. The fraction of blocks that are delayed for more than 10 minutes is roughly 8% across all configurations, too.

Our empirical analysis shows that it is the reduced frequency of Step-② events that degrades the attack effectiveness. Our log of Step-② events (i.e., missing one block) at the target shows that 1.12%, 0.59%, and 0.38% for targets with 50, 100, and 200 `maxpeers`, respectively, which makes it harder to attack them.

We find that our actual mainnet experiments above show much better attack performance than what our analytical model predicts; that is, in theory, we expect near zero probabilities of Step-② when targets are configured `maxpeers`=100 or 200, which shows a clear difference from reality. Through observation, we find that the actual number of peers that deliver unsolicited block messages (denoted n in Section IV-E) can be much smaller in practice. One reason is that a quarter of the Ethereum mainnet nodes are not able to validate (and thus relay) recent blocks because they are simply not yet synchronized [14]. Moreover, it is widely known that many protocol-deviating client implementations (such as passive monitoring supernodes [4], [26], transaction spammers [24], and conceivably sybil nodes [7]) do not dutifully participate in the Ethereum's block propagation.

B. Controlled Mainnet Experiments with Temporary Forks

Confirming the effective Gethlighting against our own nodes, we attempt to inject adversary-mined blocks to our targets. Since we do not have significant mining power for our adversarial fork experiment, we modify the adversary's mining routine so that she regularly publishes a new block every other minute (i.e., assuming 10% hash power) without a valid PoW proof. Our attack target node also skips the PoW verification for adversary's blocks.

An adversary's mining node is connected to the target with 32 Gethlighting peers to demonstrate how effectively adversary-mined blocks are accepted by a target node. Figure 9 shows that the adversary-mined blocks are accepted *immediately* by the target node when it is partitioned by our Gethlighting attack. This is because these blocks are delivered to the target in an unsolicited manner always. Note that adversary's leading cannot persist without partitioning as we assume the adversary possesses less than 50% of the mining power.

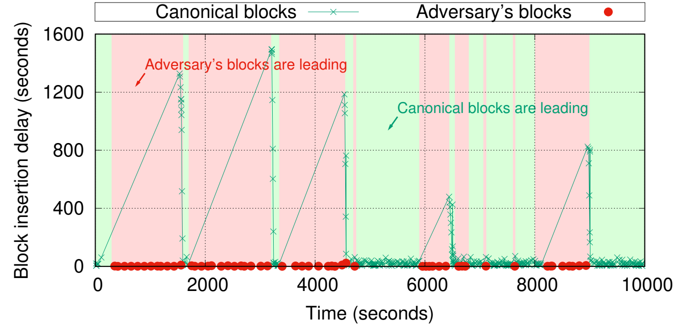


Fig. 9: Adversarial fork with Gethlighting attack. Adversary's blocks (red) are instantly accepted while canonical blocks (green) are delayed due to a TX-flooding attack.

C. Real-World Experiments in a Testnet

For our first real-world experiments, we conduct the full-scale of Gethlighting in the Rinkeby Ethereum testnet. Instead of testing TX-flooding attacks against any testnet nodes selected at random, we carefully choose and target *critical* nodes. We believe our experiment is conservative and shows the real risk in the testnet because these critical nodes are more likely to apply additional security measures than the average nodes.

Discovering critical nodes in a testnet. We are inspired by the work by Li et al. [27] as we define and search for critical Ethereum nodes. We first operate several supernodes that collect an extensive set of node addresses and client IDs of neighboring peers in both Ethereum mainnet and testnet (i.e., Rinkeby). As for the Rinkeby testnet, we discover the node addresses of additional critical nodes (i.e., bootnodes[11] and miners[12]) by matching our address collection to the known profiles of registered network nodes [1]. Total 2 RPC services, 1 miner, and 2 bootnodes are found in the Rinkeby Ethereum testnet.

Testnet results. We launch full-scale Gethlighting attacks against the discovered critical testnet nodes. We wait until at least 25 Gethlighting peers are connected to each attack target, which takes about one day or less. Then, we carry out the attacks for six hours and record the maximum block insertion delay observed in the six-hour period. Since the target nodes are not under our control, we measure the block insertion delays indirectly. That is, the block insertion time (at the target) is estimated indirectly with a `NewBlock` or `NewBlockHashes` message, as a consequence of the target's inception of a new block, received by our own Gethlighting peers. We find that the error due to our indirect measurement should be marginal with the block insertion delay measurements in the order of hundreds to thousands of seconds.

Table II summarizes our attack results. We observe that Gethlighting partitions the discovered critical nodes for more

[11] Bootstraping nodes [11].

[12] Rinkeby testnet miners are also called *signers* of the Proof-of-Authority network [36].

TABLE II: Critical testnet nodes we target in our experiments. All codenames and registered names are anonymized.

Client-*codename* (or registered name) {# of peers}, if avail	Kind	# of nodes (tested)	# of attack conns	Max block insertion delay
Geth-*o****	RPC	39 (3)	28–34	9,092–9,560 sec
Geth-*d****	RPC	1	32	2,529 sec
I*** signer	Miner	1	14	280 sec
I*** bootnode {421}	Bootnode	1	40	4,153 sec
A*** bootnode {327}	Bootnode	1	40	2,891 sec
Geth-*P****	Unknown	1	40	4,292 sec
Geth-*l****	Unknown	1	28	3,686 sec
Geth-*S****	Unknown	1	40	1,269 sec

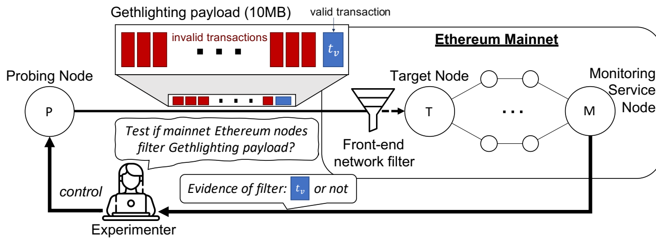


Fig. 10: Safely testing the existence of network filters for Gethlighting attack `payload` in the Ethereum mainnet. In all our mainnet experiments, the single valid transaction (t_v's) we insert into `payload` are found in the Ethereum blockchain, which shows the non-existence of such filters.

than a thousand seconds in most cases. We confirm that partitioning for more than a few hours is indeed possible. Particularly, the RPC service nodes (identified by the codename *o****) are badly damaged by Gethlighting. I*** signer node is the only experiment target we do not observe significant partitioning. It is clearly partitioned but no more than 5 minutes.

Since these target nodes are not under our control, our post-mortem analysis is, unfortunately, limited. Yet, we provide a few best guesses of ours about these testnet target nodes: (1) Many critical service nodes (e.g., codename *o****) seem to use mediocre-performing servers (thus vulnerable to Gethlighting). (2) Some critical nodes (e.g., I*** signer) appear to use a modified peer management logic, which (probably unknowingly) makes them more robust to Gethlighting. For instance, the I*** signer node tends to disconnect all peers (even benign ones) unconditionally every five minutes; hence, the average number of connections made to the node is only 14. This also shows that the targets' system performance and protocol configurations matter a lot when under Gethlighting attacks. (3) Gethlighting is also effective when a critical node is configured with a higher number of maximum peer connections. Two Rinkeby bootnodes (i.e., I*** and A*** bootnode in Table II) have more than 300 peers at the time of our experiment (which is confirmed through their dashboard where they publicly share their current numbers of peer connections [1]). The two nodes with that many peers are vulnerable to Gethlighting as well, demonstrating 4,153 sec and 2,891 sec maximum insertion delays, and this well aligns with the results of our mainnet experiment in Section V-A.

TABLE III: List of critical mainnet nodes we test.

Client-*codename*	Kind	# of nodes	Valid `txid` prefix
Geth-*E****	Miner	18	`0xe64630`
Geth-*E****	Miner	3	`0x336fae`
Geth-*E****	Miner	2	`0x9ab8c0`
Geth-*e****	Miner	1	`0x43ea7b`
Geth-*c****	Miner	1	`0xfb90a5`
Geth-*o****	RPC	63	`0xd21bf1`
Geth-*A****	RPC	18	`0x4d8d64`
Geth-*M****	RPC	6	`0x6f9017`
Geth-*a****	RPC	3	`0xe4ecd0`
Geth-*I****	Misc	1	`0x96ece0`
Geth-*k****	Misc	1	`0xd0ffe8`
Geth-*m****	Misc	1	`0xe92dd9`
Geth-*W****	Misc	1	`0x15b684`
Geth-*p****	Unknown	4	`0x0b884d`

D. Real-World Mainnet Experiments

Finally, we test the Gethlighting attack against third parties' critical nodes in the Ethereum mainnet. For obvious ethical concerns, we do *not*, however, conduct the full-scale Gethlighting attacks against the real-world mainnet nodes that are operated by others. Hence, our mainnet experiments are inherently limited.

Why mainnet critical-node experiments? So far, we have demonstrated the effectiveness of Gethlighting against our own mainnet nodes or the nodes in a testnet. However, it is insufficient to show the feasibility of the attack against the real-world mainnet targets because the service operators of critical real-world mainnet nodes may have deployed a front-end network filter (e.g., intrusion detection or prevention system [28]) that removes our attack payloads (i.e., TX-flooding messages). To rule out these possibilities of such network filters and confirm the feasibility of Gethlighting in the mainnet, we test whether the Gethlighting attack payloads successfully reach real-world critical Ethereum nodes in the mainnet; see below for more details.

Discovering critical nodes in the mainnet. As we explain earlier in this section, we operate our own supernodes to discover critical nodes in the mainnet. We identify more than 25K unique peers in the mainnet for a month. We discover 4 RPC services and 5 mining nodes in the Ethereum mainnet by analyzing the client codenames we collect [27].

Safely testing the existence of filters in the mainnet. We devise the following probing technique to test whether our Gethlighting attack payloads reliably reach real-world target nodes in the mainnet or are *filtered out* by a front-end security system. In Figure 10, a probing node P (e.g., a customized Geth client) establishes a single outgoing peer connection to a mainnet target T of interest. P sends only a single attack payload message (denoted as `payload`) that consists of 96,000 invalid transactions and one *valid* transaction t_v at the end. Our probing node P never disseminates t_v via any other messages; therefore, if t_v is observed via a monitoring service node M in the Ethereum's canonical blocks, we can confirm that our target T has received and properly processed `payload`.

Table III summarizes our probing experiment results. We observe that our probing nodes are always able to send Gethlighting attack payloads (i.e., `payload` messages) to these

14 critical nodes in the mainnet, and we find the provided valid transactions (i.e., t_v) are always included in the Ethereum blockchain. This provides a clear evidence that those critical mainnet nodes do not have any specific network filters that prevent Gethlighting.

Note that we send only a single attack payload to a real-world target (unlike our full-scale Gethlighting attacks) and this does not cause any noticeable burden to the target.

E. Ethical Concerns

Our attack experiments are conducted with great care to avoid any disruption to the operation of the Ethereum mainnet. First, in our experiments in Section V-A and Section V-B, we launch attacks against our own Ethereum client nodes. Thus, in these experiments, we never send any attack messages (e.g., TX-flooding and adversary-mined block messages designed for our attack) to any Ethereum mainnet nodes other than our own target. We ensure this by restricting our attack nodes to make connections only to our target nodes. Our target nodes are also set to not relay any Ethereum messages to other benign peers in the mainnet. This way, we conduct highly isolated experiments with our nodes while they are connected to the mainnet.

In our real-world experiments in the Ethereum testnets in Section V-C, we actually create damage to some critical nodes (not our own nodes) in the testnets but limit the total attack execution to six hours and avoid repeated experiments. We believe that a certain level of attack demonstration in blockchain testnets for research purposes is widely accepted in the security community [27].

Last, in our real-world experiments in the mainnet, we only test the reachability of Gethlighting attack payloads to real-world mainnet target nodes but never cause any damage to the targets.

VI. Attack Scalability and Cost

Since Ethereum is a permissionless public blockchain, an adversary can *reuse* the same set of virtual machines (VMs) to establish multiple concurrent peer connections to multiple target nodes. Simple modifications to Ethereum clients (e.g., increasing the `maxpeers` setting) enable the adversary's VMs to create an unlimited number of potential target nodes, as long as the VMs' resources support the attack operation.

From our experiments, we learn that when an adversary exploits 25 peer connections, a single adversary peer generates approximately 7 Mbps attack traffic. Because VMs used for TX-flooding do not need to be fully-functioning nodes, bandwidth becomes a limiting factor when reusing VMs for TX-flooding. To simplify the discussion, assume that we use 25 distinct VMs to establish 25 peer connections to a single target node. We confirm via our own `iperf` experiment that a `t2.large` Amazon EC2 instance supports 1 Gbps upstream bandwidth. Thus, one low-cost VM can target approximately 140 nodes concurrently.

We use the Amazon EC2 pricing table for the `t2.large` instance (i.e., $0.0928 USD per hour, $0.09 USD per GB) and consider the attack is executed for 24 hours. In total, it would cost only about $5,714 for targeting all 10K full nodes in Ethereum for 24 hours.

VII. Potential Countermeasures

As we already discussed in Section IV, the Gethlighting partitioning attack is enabled because of the five Ethereum characteristics **[EC1]–[EC5]**. Since many of these Ethereum characteristics are the result of some fundamental trade-offs in the system design, simply disabling or removing them does not make a reliable countermeasure against Gethlighting. Instead, we offer several potential smaller tweaks to the existing Ethereum protocols (§VII-A–§VII-E) and one operational practice that mitigates the Gethlighting attack. We summarize these potential countermeasures in Table IV

Existing *anti-eclipse* schemes do *not* work. Before we delve into the potential countermeasures, let us clarify why existing anti-eclipse schemes do not work against Gethlighting at all. Existing eclipse attacks and their variants [3], [22], [23], [29], [37] have led to several practical countermeasures against them. One highly effective family of techniques is to ensure a small number of reliable, benign peer connections. Such connections can be trusted nodes [23], a centralized block propagation network [16], [18], or a hijacking-resistant relay node [2]. All these defense schemes do not mitigate Gethlighting at all because Gethlighting does not require a total monopoly of peer connections in the first place.

Another practical countermeasure is to make a few additional outbound peer connections for better connectivity. This has been adopted for mitigating eclipse and Erebus attacks [22], [37]. As we show in Section V-A and V-C, any minor increase in the number of peer connections does not make an effective countermeasure against Gethlighting.

A. Taming TX-flooding

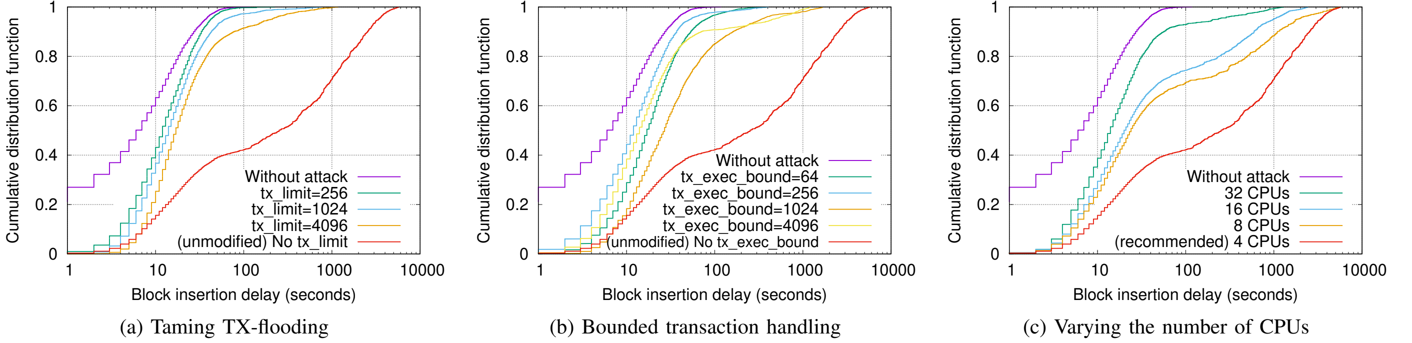
In Ethereum, both blocks and transactions can be propagated either solicitedly or unsolicitedly. Gethlighting's TX-flooding strategy exploits the unsolicited mode of transaction delivery.[13] Disabling the unsolicited delivery mode for transactions (i.e., new transactions should be pooled by explicit requests) would make an effective countermeasure against the TX-flooding strategy. Note that Bitcoin by design disallow unsolicited delivery of transactions and thus is robust against TX-flooding attacks.

A much less intrusive change to the Ethereum protocol would be to simply limit the number of transactions that can be delivered by a single `Transactions` message. The maximum number of transactions in Ethereum is only limited by the maximum size of a `Transactions` message (i.e., 10 MB). A further limitation can effectively reduce the impact of weak per-peer isolation by decreasing the processing time bound s in Section IV-A. Figure 11(a) shows how `tx_limit` (i.e., the number of transactions that can be informed with a single `Transactions` message) affects the attack performance of the TX flooding strategy. The block insertion delays are measured for every block during a 24-hour-long attack experiment. There have been only a few blocks with insertion delays larger than two minutes when `tx_limit`=256. However, still 1% of the blocks have experienced insertion delays larger than five minutes with `tx_limit`=1024.

[13] Gethlighting's choice of unsolicited mode for *transactions* should not be confused with the use of solicited mode for *block* propagation.

TABLE IV: Summary of countermeasures, affected attack steps, and their side effects

Countermeasures	Corresponding Ethereum's Characteristics	Related Attack Steps	Disadvantages or Side Effect
Taming TX-flooding	**[EC1]**	Step-② & Step-③	Significant changes to P2P
Bounded transaction handling	**[EC1]**	Step-② & Step-③	A few worst-case delays, slow TX handling
Making clients block-synchronized first	**[EC1]**	Step-② & Step-③	Weak transaction propagation
Gethlighting-resistant block propagation	**[EC2]**, **[EC5]**	Step-②	Msg complexity, risk of DoS
Banning Gethlighting peers	**[EC3]**	Step-② & Step-③	False positives, risk of new attacks
Overprovisioning clients	**[EC1]**	Step-② & Step-③	Expensive



(a) Taming TX-flooding

(b) Bounded transaction handling

(c) Varying the number of CPUs

Fig. 11: Effectiveness of countermeasures.

Although rate limiting transactions is less intrusive than the other alternative, it still requires some non-trivial changes to the existing protocol. Therefore, special care must be taken before applying this change to the Ethereum network protocol.

B. Bounded Transaction Handling

We learn in Section IV-A that the upper-bound of a peer's service rate, due to TX-flooding, is inversely proportional to the system parameter s (i.e., the time quantum of Go's preemptive scheduling). Thus, reducing s can make an effective countermeasure as it increases the service rate of benign peers, shortens message queues, and decreases delays.

One caveat, however, is that a system-wide reduction of Go's scheduling time quantum (i.e., 20 msec) could potentially result in performance degradation or unexpected problems. To mitigate any such unforeseen problems, we apply a decreased time bound exclusively for the TX handling routine. We modify the transaction validation routine so that the goroutine, which executes it, should yield to other goroutines after validating every `tx_exec_bound` transactions.[14] This way, Gethlighting peers would not be able to continuously occupy the entire allocated time quantum of 20 msec. The Gethlighting-induced job of validating transactions is then interleaved to allow the execution of message handling goroutines of benign peers.

Figure 11(b) shows how `tx_exec_bound` (i.e., the maximum number of transactions that can be handled before yielding to other goroutines) affects the attack performance. It is generally an effective countermeasure to Gethlighting but one downside is that it does not completely address the attacks; see the result with `tx_exec_bound`=64 is still worse than the 'no attack' case. Compared to limiting the transaction message size, bounding transaction handling can be preferred in practice since it does not require any changes to the protocol.

C. Making Clients Block-Synchronized First

One simple-yet-effective countermeasure against Gethlighting is to prioritize any protocol messages exchanged for synchronization (e.g., informing a block, announcing a block hash, requesting or responding with block header or body) over other messages, such as transaction messages. The rationale behind this strict prioritization is that a good synchronization of blocks must precede transactions. When a node is not synchronized with the canonical chain, transactions cannot be correctly validated.

To test this, we implement the message prioritization architecture, which augments the existing single protocol message channel[15] to high- and low-priority channels. The protocol messages exchanged for synchronization are handled with a strictly higher priority. We run a TX-flooding attack with 40 Gethlighting peers against our own Ethereum full-node that is configured as recommended [13] (i.e., four CPU cores and 16 MB memory). No partitioning event lasts longer than three minutes for a 24-hour-long experimentation.

Note, however, that such a strict prioritization based on message types can potentially create new problems and even open up a new avenue of attacks and therefore should be carefully reviewed and studied before any deployment. For example, transaction propagation may become slower, rendering Ethereum slower to accept transactions. Worse, some

[14] Go runtime's `Gosched()` is used to yield to other goroutines. [19]

[15] See Figure 2 in Section II-B for more information.

transactions can be targeted and dropped by adversaries who flood more higher-priority messages in the network.

D. Gethlighting-Resistant Block Propagation

As a mitigation of Gethlighting in Step-②, we may sacrifice the high optimization in the block propagation protocol a little by introducing some redundancy and concurrent block propagation. One possible yet-radical solution is to prevent a peer from receiving a new block in a solicited manner. This ensures that Step-② never succeeds as the solicited block propagation is not needed in the first place. The price of this mitigation is the significantly increased message complexity for block propagation.

E. Banning Gethlighting Peers

One convenient and effective countermeasure is to ban peers that exhibit potentially malicious behaviors; e.g., flooding with invalid-looking transactions. As we discuss in Section IV-C, the current Ethereum clients do not have such peer banning mechanisms. One possible peer banning scheme is to ban peers that send too many invalid-looking transactions. This banning scheme can trivially disable the Gethlighting attacks since Gethlighting adversaries cannot maintain their attack effectiveness when all her peers get disconnected from the target node as soon as they perform any attack strategies.

However, there are potential side effects that are not trivial to address. First, Ethereum nodes may ban peers by mistakes. For example, a peer may get banned mistakenly for sending invalid-looking transactions when it has sent transactions that look perfectly valid from its own perspective. This may arise due to the lack of synchronization between two peers. Or, a peer may get disconnected for not responding in a timely fashion when it is actually experiencing network failures or delays. Second, malicious parties may attempt to exploit these new peer banning schemes. With these schemes, a simple denial-of-service (DoS) attack against a target node can easily disconnect all of the target's peers, which can be used as stepping stones for other attacks [3], [22], [37]. Or, creating a temporal chain inconsistency in the network may cause sudden bulk disconnections of network peers.

F. Overprovisioning Clients

One side-effect-free countermeasure against the TX-flooding strategy is to provision more resources to a target node. The more CPU cores are allocated to a target, the less time is expected to handle each protocol message (see Section IV-A). Thus, TX flooding attacks become less effective when targeting an overprovisioned target node.

Figure 11(c) shows how the number of CPUs at a target node affects the effectiveness of TX-flooding attacks. Having more CPU cores clearly lowers attack effectiveness. When a target has four CPU cores (as recommended by the official Ethereum client [13]), the median partitioning duration is about 250 seconds. This drastically decreases to around 25 seconds when we overprovision the target with the twice the recommended number of CPU cores.

However, the result also shows that overprovisioning cannot be an ultimate solution to Gethlighting. Even when a target has $2\times$ (i.e., 8 CPUs) or $4\times$ (16 CPUs) of the recommended number of CPUs, the worst case partitioning durations can easily go above 1,000 seconds (see their 90-th percentile in Figure 11(b)). Moreover, it is not a cost-effective solution since only marginal reduction of partitioning duration is expected as we keep doubling the number of CPUs at the target.

VIII. Discussion

A. Opportunistic Attacks

While the Gethlighting attack has notable advantages mentioned above, it has one minor limitation. That is, the attack is opportunistic in terms of the duration of the partitioning process and the time required to begin partitioning a target. This is because its attack strategies are dependent not only on the adversary's connections but also on benign peer connections, which account for roughly half of the target's peer connections. In practice, therefore, adversaries would need to wait for a short period of time (e.g., a few minutes) before initiating partitioning, and then attempt a number of partitioning tries to generate a partitioning event lasting long enough to accomplish their ultimate goals (e.g., double spending). Gethlighting-induced partitioning is effective intermittently since it introduces mild disruption in processing received blocks, often enough to trigger timeouts in message handling. Some messages received during the bulk block download (i.e., Step-③) occasionally may get processed in a timely manner, rendering a successful recovery from partitioning. The general trend, nonetheless, is that the more Gethlighting peers, the longer partitioning is (Section V-A).

B. Detection of Gethlighting

Partitioning detection mechanisms [2] against existing attacks [3], [22], [33], [37] would determine whether or not a target node is eclipsed (i.e., attackers control all peer connections). As a result, these measures would never detect Gethlighting, as the attack never eclipses any Ethereum node.

However, Gethlighting is not imperceptible. Gethlighting peers behave differently from most benign peers and, thus, in theory, they can be detected with proper measures. Given the fact that Gethlighting is not considered a volumetric attack, it is difficult to detect at the network layer (e.g., IDS or firewall). Thus, each node in the system should identify Gethlighting peers at the application layer, which would necessitate some adjustment of Ethereum client applications.

IX. Related Work

A. Partitioning Attacks in Blockchain

Most known partitioning attacks aim to *monopolize* a target's peer connections completely. The Eclipse attacks against Bitcoin [22] and Ethereum [23], [29] demonstrate that an adversary can monopolize all outgoing peer connections by exploiting software vulnerabilities, which have been patched since their disclosure. Additionally, it is also demonstrated that a strong network adversary (e.g., malicious ISP) can launch a man-in-the-middle attack, gaining complete control of a target node's peer connections via a data-plane [37] or control-plane attacks [3]. An Eclipse attack against the InterPlanetary File System (IPFS), which uses a peer-to-peer networking

TABLE V: Comparison with existing blockchain partitioning attacks

Attack	Target blockchain	Attack requirements				Attack characteristics		
		# attack conns	Network resource	Attack preparation	Need reboot?	Partitioning duration	Attack fork	Deter-ministic?
Heilman *et al.,* 2015 [22]	Bitcoin	All	$\sim$4.6K bots	Several hours	○	Permanent*	○	○*
Wüst and Gervais 2016 [42]	Ethereum	1	1 IP	Several weeks†	×	Permanent	×$^{\diamond}$	○
Apostolaki *et al.,* 2017 [3]	Bitcoin	All	Hijacking $<$100 BGP prefixes	$\sim$90 seconds	×	Permanent*	○	○*
Marcus *et al.,* 2018 [29]	Ethereum	All	2 IPs	Several minutes$^{\bullet}$	○	Permanent	○	○
Henningsen *et al.,* 2019 [23]	Ethereum	All	2 IPs	Several days	×	Permanent	○	○
Tran *et al.,* 2020 [37]	Bitcoin	All	Compromising few large ISPs	Several weeks	×	Permanent*	○	○*
Saad *et al.,* 2021 [33]	Bitcoin	All	$\sim$100 IPs	Not evaluated	○‡	Not evaluated*	○	×
Gethlighting	Ethereum	$\sim$25	1 IP	$<$1 day$^{\triangledown}$	×	$\sim$3 hours	○	×

$\star$: not anymore with the deployed peer eviction mechanism $\dagger$: for adversarial chain generation $\ddagger$: partitioning newly arriving targets only
$*$: depending on network churn $\diamond$: only from the genesis block $\bullet$: to craft node IDs $\triangledown$: based on testnet evaluation

architecture similar to blockchain systems, is also presented recently [32]. Our Gethlighting attack significantly reduces the attack requirement (i.e., from complete control of peer connections to partial control), rendering any existing detection mechanisms or countermeasures designed for existing partitioning attacks (e.g., routing-aware peering [38], SABRE [2]) completely ineffective.

Earlier in 2016, Wüst and Gervais [42] present a new permanent Ethereum partitioning attack with only a single malicious connection to a target node. The Wüst-Gervais (or WG) attack inserts an adversary-generated chain that is longer than the canonical chain to stop the target's chain growth, whereas the Gethlighting attack triggers a subtle condition in the Geth's multi-peer connection scheduling mechanism to time out some blocks at the target. Although Gethlighting requires more malicious peer connections than the WG attack needs, occupying some more (e.g., 25) connections of the target for Gethlighting costs very little in practice (see Section III-A). In contrast, generating an alternative chain from the genesis block for the WG attack would cost significantly, which is completely unnecessary for Gethlighting. Last, the WG attack was addressed with a hotfix [35] that prevents forking at the genesis block. Gethlighting, however, is less likely to be fixed, completely and without any side-effect (see Section VII), by a single hotfix because the attack is made possible due to several system and protocol characteristics (as we summarize as **[EC1]–[EC5]** in Section IV).

Table V summarizes our comparison of Gethlighting with the existing blockchain partitioning attacks. The attack requirements are compared in terms of the required number of attack connections to be monopolized, required network resource, attack preparation time, and whether the target reboot is necessary for an attack. We also compare the maximum partitioning duration, whether the attacker can inject adversary-mined branch at an arbitrary block height, and whether the attack is deterministic (Section VIII-A). Gethlighting is the first partitioning attack that does not require a total eclipse of peer connections since an earlier attack by Wüst and Gervais in 2016, which has been prevented after a simple fix [35]. Gethlighting also does not require multiple IP addresses and can target any existing reachable nodes without requiring the target to restart.

TABLE VI: Gethlighting vs. DETER [27] Differences

Characteristics	Gethlighting	DETER
Attack goal	Delay *block* insertion	Drop some *transactions*
Consequence	Target node is *partitioned* (i.e., *safety violation*)	Transactions are *censored*
Implication	$\not\Rightarrow$ DETER	$\not\Rightarrow$ Gethlighting
Payload	Invalid transactions	Invalid* transactions (* carefully-crafted)
Attack cost	A few servers; zero gas	One server; zero gas
Detection	Difficult at network; Feasible at application	Difficult at network; Feasible at application

B. Denial-of-Service Attacks in Blockchain

Availability attacks have been studied in the literature because the *liveness* of a blockchain system is a critical property. Baqer et al. [5] present an empirical study on a Bitcoin stress test campaign that has resulted in a DoS attack. Mirkin et al. [30] claim that adversaries can halt PoW blockchain systems by publishing only partial information (e.g., header) of her mined blocks, with fewer resources (e.g., 21% of the total hash power). Vasek et al. [39] conduct an investigation into known DDoS attacks and assess the anti-DDoS system adoption status of the Bitcoin ecosystem's key operational entities (e.g., cryptocurrency exchanges and mining pools).

A recent work by Li et al. [27], called a DETER attack, shares some similarity to Gethlighting as both present new low-rate denial-of-service attacks in Ethereum. However, the two attacks are entirely different and thus should not be confused with each other. The main attack goal of DETER is to drop some legitimate *transactions* at a target node and thus transactions in Ethereum can be censored by the adversary. In contrast with DETER, Gethlighting aims to delay *block* insertion at a target node so significantly that the target is virtually partitioned from the Ethereum network, rendering the violation of the safety property. Also, neither Gethlighting nor DETER imply the successful execution of the other. One characteristic that may make the two attacks look similar is their attack payload — i.e., both attacks send invalid transactions to their targets. Yet, their strategies to creating invalid transactions

differ; e.g., DETER requires more carefully created invalid transactions. In short, the Gethlighting attack is different from DETER as it does not share the goals or the attack techniques of DETER. Table VI summarizes the key differences between Gethlighting and DETER.

X. Conclusion

In this work, we show that a subtle (seemingly insignificant) low-rate denial-of-service vulnerability in blockchain peer-to-peer scheduling algorithms can lead to a significant network-wide partitioning vulnerability. Worse yet, attacks like Gethlighting renders all existing anti-eclipse defenses ineffective because they do not eclipse their target nodes in the first place. Our hotfix to Gethlighting has been included in the upcoming Geth release and we leave the generalization of this new attack vector in other blockchain networks (including Ethereum 2.0) for future work.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. We also thank Yongdae Kim for useful comments on an early version of the paper. This work was supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government [23ZR1300, Research on Intelligent Cyber Security and Trust Infra].

References

[1] Rinkeby: Network dashboard. https://www.rinkeby.io/#stats.

[2] M. Apostolaki, G. Marti, J. Müller, and L. Vanbever, "Sabre: Protecting bitcoin against routing attacks," in *Proc. NDSS*, 2018.

[3] M. Apostolaki, A. Zohar, and L. Vanbever, "Hijacking bitcoin: Routing attacks on cryptocurrencies," in *Proc. IEEE S&P*, 2017.

[4] S. Baek, H. Nam, Y. Oh, M. Tran, and M. S. Kang, "On the claims of weak block synchronization in bitcoin," in *Proc. International Conference on Financial Cryptography and Data Security (FC)*, 2022.

[5] K. Baqer, D. Y. Huang, D. McCoy, and N. Weaver, "Stressing out: Bitcoin "stress testing"," in *Proc. International Conference on Financial Cryptography and Data Security (FC)*, 2016.

[6] Bitcoin Core PR Review Club, "Add unit testing of node eviction logic," 2020, https://bitcoincore.reviews/20477.

[7] J.-P. Eisenbarth, T. Cholez, and O. Perrin, "Ethereum's peer-to-peer network monitoring and sybil attack prevention," *Journal of Network and Systems Management*, 2022.

[8] Ethereum, "Node discovery protocol v5," 2020, https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md.

[9] Ethereum Core Developers, "eth: implement the newblockhashes protocol proposal," 2015, https://github.com/ethereum/go-ethereum/pull/1188.

[10] Ethereum Foundation, "Light ethereum subprotocol," 2021, https://github.com/ethereum/devp2p/blob/master/caps/les.md.

[11] ——, "Connecting to the network," 2022, https://geth.ethereum.org/docs/fundamentals/peer-to-peer.

[12] ——, "Ethereum wire protocol," 2022, https://github.com/ethereum/devp2p/blob/master/caps/eth.md.

[13] ——, "Node and clients," 2022, https://ethereum.org/en/developers/docs/nodes-and-clients/.

[14] Ethernodes. The ethereum node network & node explorer. https://ethernodes.org.

[15] Ethstats. Ethereum network status. https://ethstats.dev.

[16] Falcon, "A Fast Bitcoin Backbone," 2016, https://www.falcon-net.org/.

[17] W. Fan, S. Wuthier, H.-J. Hong, X. Zhou, Y. Bai, and S.-Y. Chang, "The security investigation of ban score and misbehavior tracking in bitcoin network," in *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2022.

[18] FIBRE, "Fast Internet Bitcoin Relay Engine," 2020, http://bitcoinfibre.org/.

[19] Go. Go runtime package. https://pkg.go.dev/runtime.

[20] Go Ethereum github, "1.11.0 milestone," 2022, https://github.com/ethereum/go-ethereum/milestone/136.

[21] Google, "pprof is a tool for visualization and analysis of profiling data." 2016, https://github.com/google/pprof.

[22] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on bitcoin's peer-to-peer network," in *Proc. USENIX Security*, 2015.

[23] S. Henningsen, D. Teunis, M. Florian, and B. Scheuermann, "Eclipsing ethereum peers with false friends," *arXiv preprint arXiv:1908.10141*, 2019.

[24] H. Heo and S. Shin, "Behind block explorers: Public blockchain measurement and security implication," in *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2021.

[25] N. Johnson, "Discussion of square-rooted block propagation in ethereum@allcoredevs," 2018, https://gitter.im/ethereum/AllCoreDevs?at=5b9e20db54587954f9ad87bd.

[26] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey, "Measuring ethereum network peers," in *Proc. ACM Internet Measurement Conference (IMC)*, 2018.

[27] K. Li, Y. Wang, and Y. Tang, "Deter: Denial of ethereum txpool services," in *Proc. ACM CCS*, 2021.

[28] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *Journal of Network and Computer Applications*, 2013.

[29] Y. Marcus, E. Heilman, and S. Goldberg, "Low-resource eclipse attacks on ethereum's peer-to-peer network." *IACR Cryptol. ePrint Arch.*, 2018.

[30] M. Mirkin, Y. Ji, J. Pang, A. Klages-Mundt, I. Eyal, and A. Juels, "Bdos: Blockchain denial-of-service," in *Proc. ACM CCS*, 2020.

[31] K. Nayak, S. Kumar, A. Miller, and E. Shi, "Stubborn mining: Generalizing selfish mining and combining with an eclipse attack," in *Proc. IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016.

[32] B. Prünster, A. Marsalek, and T. Zefferer, "Total eclipse of the heart–disrupting the {InterPlanetary} file system," in *Proc. USENIX Security*, 2022.

[33] M. Saad, S. Chen, and D. Mohaisen, "SyncAttack: Double-spending in Bitcoin Without Mining Power," in *Proc. ACM CCS*, 2021.

[34] M. H. Swende, "eth/fetcher: throttle peers which deliver many invalid transactions," 2022, https://github.com/ethereum/go-ethereum/pull/25573.

[35] P. Szilágyi, "eth/downloader: bound fork ancestry and allow heavy short forks," 2016, https://git.with.parts/mirror/go-ethereum/commit/39ce85cf5d119ef830561ecdc4096bfe565bc5c1.

[36] ——, "Eip-225: Clique proof-of-authority consensus protocol," 2017, https://eips.ethereum.org/EIPS/eip-225.

[37] M. Tran, I. Choi, G. J. Moon, A. V. Vu, and M. S. Kang, "A stealthier partitioning attack against bitcoin peer-to-peer network," in *Proc. IEEE S&P*, 2020.

[38] M. Tran, A. Shenoi, and M. S. Kang, "On the Routing-Aware Peering against Network-Eclipse Attacks in Bitcoin," in *Proc. USENIX Security*, 2021.

[39] M. Vasek, M. Thornton, and T. Moore, "Empirical analysis of denial-of-service attacks in the bitcoin ecosystem," in *Proc. International Conference on Financial Cryptography and Data Security (FC)*, 2014.

[40] M. V. D. Wijden, "Mark ddos attacker as bad in p2p layer," 2021, https://github.com/ethereum/go-ethereum/pull/22198.

[41] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[42] K. Wüst and A. Gervais, "Ethereum eclipse attacks," ETH Zurich, Tech. Rep., 2016.

APPENDIX

A. Simple Tests with OpenEthereum

In this section, we demonstrate the attack feasibility of Gethlighting against OpenEthereum, the second most popular Ethereum client. The attack target is our Ethereum full-node running OpenEthereum v3.3.3 on Amazon EC2's `i3.8xlarge` instance. Note that Our Gethlighting implementation based on Geth 1.10.20 (Section V) is used without any modification since they both communicate with each other in the Ethereum protocol. In overall, a TX-flooding attack with 25 attack nodes is launched for 5 hours to show the longest partitioning duration of 2,004 seconds.

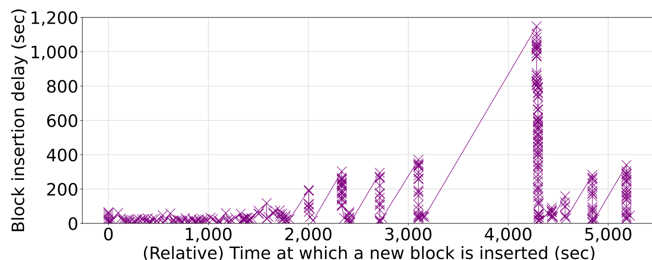


Fig. 12: An example snapshot of an OpenEthereum node under the Gethlighting attack.

Figure 12 shows the snapshot of the first 5,200 seconds of the target node under a Gethlighting attack. We observe periodic partitionings up to 1,148 seconds as soon as 18 Gethlighting peers are connected (at 1,800 sec) to the victim. Even worse, a majority of benign peer connections get disconnected at the onset of each partitioning; that is, *all* benign peer connections get disconnected by the target during 5 out of 7 partitioning episodes of Figure 12. The reasons of massive disconnection turn out to be timer expiration of solicited messages: a request for block header is not responded in 3 seconds (`FORK_HEADER_TIMEOUT`); a request for a transaction is not answered in 10 seconds (`POOLED_TRANSACTION_TIMEOUT`); and a block body retrieval is not fulfilled in 20 seconds (`BODIES_TIMEOUT`). OpenEthereum not only applies shorter hard-coded timeout values than Geth, but also disconnects remote peers if any of the timers expire. The consequence is catastrophic; a target loses *all* peer connections, and further implying that Gethlighting is a very effective stepping stone for other attacks [3], [22], [37].

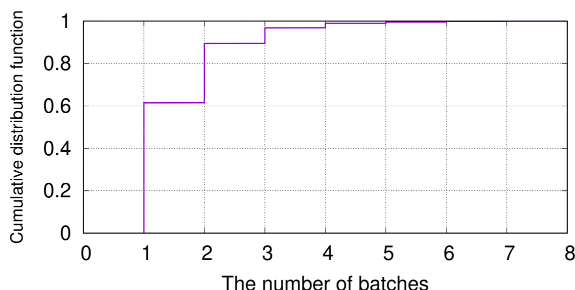


Fig. 13: Distribution of the number of block announcement batches.

B. Block Announcement Batches

Figure 13 depicts the distribution of measured block announcement batches from one of our 24-hour Gethlighting attack experiments. Block announcement messages arriving in a 0.4-second time window are merged into a single batch. The number of batches is one or two more than 95 percent of the time. This confirms that a Gethlighting target has only one or two chances to solicitedly retrieve a new block as we discussed in Section IV-B.

C. Why are transactions exceptionally more effective than other message types?

For successful Gethlighting, the attack must continuously occupy the full span of message handling goroutine's time slot (i.e., 20 msec) every time it is scheduled for execution (Section IV-A). Gethlighting with a large number of invalid transactions can accomplish this because of Ethereum's gracious handling of transactions (Section IV-C). That is, every delivered transaction is faithfully processed one-by-one, rendering consecutive full occupation of execution time slots (as depicted in Figure 4).

Other message types do not achieve this because they either involve IO operations that make the goroutine get preempted or the entire delivered data cannot occupy long enough CPU time by early short-circuiting. For example, a `NewBlock` message containing a block with thousands of transactions is immediately discarded due to invalid proof-of-work and thus induces only few CPU cycles for a hash computation.

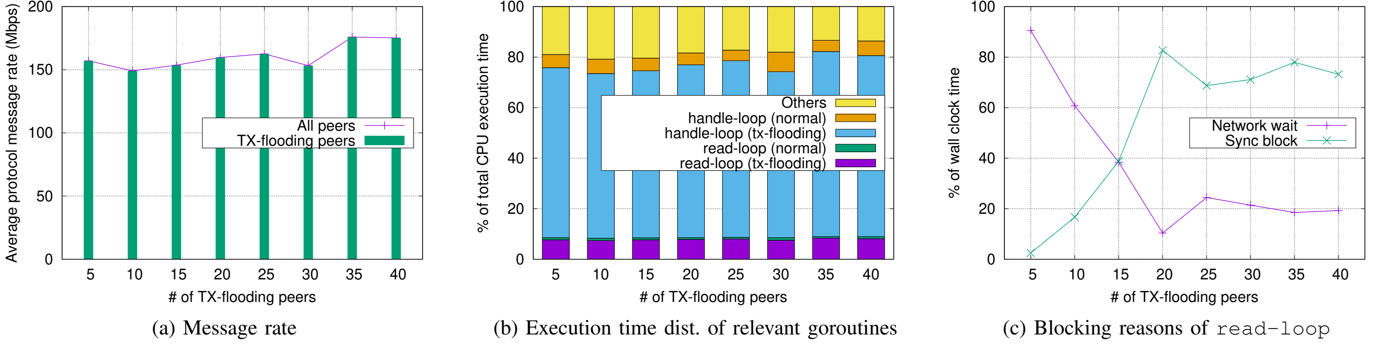
D. Why are messages delayed longer with more TX-flooding peers?

Here, we extend our evaluation of the controlled experiment in Section V-A to better understand the inner-workings of a target Ethereum node under Gethlighting attacks via goroutine profiling. A Go profiling tool `pprof` [21] is enabled at the attack target Geth process to build goroutine execution profiles from actual execution trace log.

Message handling goroutines. For each peer connection, two goroutines (lightweight threads supported by Go) are in charge of handling incoming peer messages (Figure 2): the P2P message handler reads a peer message from the network and sends it to a subprotocol Go channel (e.g., ETH), while the ETH message handler reads a message from the channel and processes it.

The pseudo-code for the relevant Geth functions is shown in Algorithm 1 and the functions are executed by two peer message handling goroutines: `read-loop` goroutine (i.e., P2P message handler in Figure 2) executing `readLoop()` and `handle-loop` goroutine (i.e., ETH message handler) executing `ETH.handleLoop()`. Every time a new peer connection is established, both goroutines are invoked. The separation of peer message handling is done for the sake of modularity, as an Ethereum node can communicate in subprotocols other than the default Ethereum protocol (e.g., snap). The Go channel between goroutines is an *unbuffered* channel that can only hold one single message at a time.

Invariable attack traffic and CPU execution times. Because our TX-flooding procedure transmits back-to-back 10 MB



(a) Message rate

(b) Execution time dist. of relevant goroutines

(c) Blocking reasons of `read-loop`

Fig. 14: Controlled experiment results.

Algorithm 1 Geth peer message handling functions

```
/* P2P peer message handling loop                                  */
Func readLoop()
    while true do
        msg ← readMsg()
        proto ← getProto(msg.code)
        /* proto.in is unbuffered input channel of                 */
        /* matching subprotocol                                    */
        send msg to proto.in
    end
end

/* ETH subprotocol messsge handling loop                           */
Func ETH.handleLoop()
    while true do
        /* wait until receiving a message                          */
        /* from the input channel                                  */
        receive msg from ETH.in
        ETH.handle(msg)
    end
end
```

messages, it is reasonable to assume that more peers will be able to deliver more messages. However, as shown in Figure 14(a), the average message rate does not increase as the number of Gethlighting peers increases. The TCP flow control mechanism does not allow an application to send data bytes faster than the rate at which data is consumed at the other end of the pipe in the long run. It appears that the amount of inbound traffic indeed reflects the target Geth process's CPU resources. However, a TCP sender can temporarily push data up to the buffer size (i.e., TCP receive window size) unless network congestion is a bottleneck.

Figure 14(b) demonstrates that the CPU resource distribution between normal and Gethlighting peers is also invariant; in other words, an adversary does not simply force a target node to allocate more CPU resources for her messages by increasing the number of Gethlighting peer connections. It is worth noting that the invariability of CPU execution time distribution corresponds to the observed constant attack traffic. Attack traffic should have increased if more CPU time was allocated to Gethlighting peers' goroutines.

Incoming message queuing due to sluggish message processing. When there are 20 or more TX-flooding peers, we find that lagging in message processing becomes the dominant reason for blocking read-loop goroutines from execution. Figure 14(c) depicts the average proportion of time that normal peers' `read-loop` goroutines stop execution for two reasons: *network wait* is a blocking reason caused by a goroutine waiting for new data bytes to arrive from a network pipe, while *sync block* indicates that a message in the subprotocol Go channel has not yet been pulled by the consumer goroutine at the other end of the channel. If there are only 5 Gethlighting peers, a normal peer's `read-loop` spends 90% of its time waiting for a new message from the remote peer. On the other hand, there is almost no stopping (i.e., *sync block*) due to any delayed message processing by the subprotocol handler. However, if there are 20 or more Gethlighting peers, the `read-loop` goroutine spends more than 70% of its running time waiting for a message in the channel to be pulled; as a result, newly arrived network data bytes (in the TCP receive buffer) cannot be pulled by `read-loop`, rendering burst incoming messages (stacked up at the buffer) waiting for the application to consume. This effectively renders an *accumulation effect of message processing delays*.

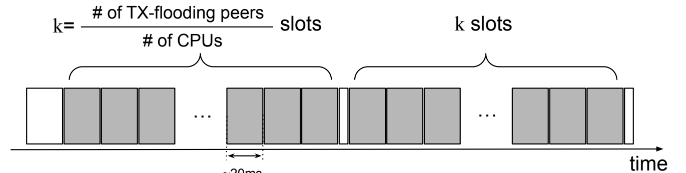


Fig. 15: Interleaved execution of `handle-loop` goroutines. The execution of a normal peer's `handle-loop` (white) is interleaved by k successive execution slots of the Gethlighting peers (grey).

Interleaved execution of goroutines. Handling Ethereum messages (via `handle-loop`) is a CPU-intensive process that reserves CPU execution time based on which tasks should be performed with the message. For example, a message informing a newly mined block consumes far more CPU time than a message propagating a single transaction. The `handle-loop` goroutine for each peer connection is either preempted voluntarily (i.e., blocking wait for a message in the Go channel) or by a non-cooperative preemption by the scheduler (i.e., outstanding Ethereum message processing is incomplete). The current (i.e., Go 1.17.8) scheduler preempts a goroutine that is running for more than the predefined time slot of 20 msec.

The Go scheduler maintains a FIFO (First-In-First-Out) local queue of runnable goroutines for each CPU core, resulting in an interleaved execution pattern of goroutines. That is, a preempted (or voluntarily scheduled-out) goroutine can only be scheduled-in after all other runnable goroutines in the queue have been executed. This execution pattern is depicted in Figure 15. It is worth noting that k (the number of consecutive execution slots occupied by goroutines handling TX-flooding messages) is a reciprocal of the number of CPUs and proportional to the number of Gethlighting peers.

E. Occupying Target's Peer Connections

We have logged all peer join/leave events of an Ethereum mainnet node by keeping the node online for a month. We simulate how many (benign) peer connections can be sustained using the event logs, assuming that the connection slots that become available (as short-lived connections leave) are immediately occupied by the adversary.

Figure 16 shows our simulation results, showing how the number of peer connections decay as the attacker begins from two hours to two weeks after the target node goes online. Almost all incoming connection slots (i.e., $50 \times \frac{2}{3} \approx 33$) of a two-hour-old node become unoccupied within only a few hours; however, a node that has been online for two weeks can sustain a half of the total connections for more than ten days.

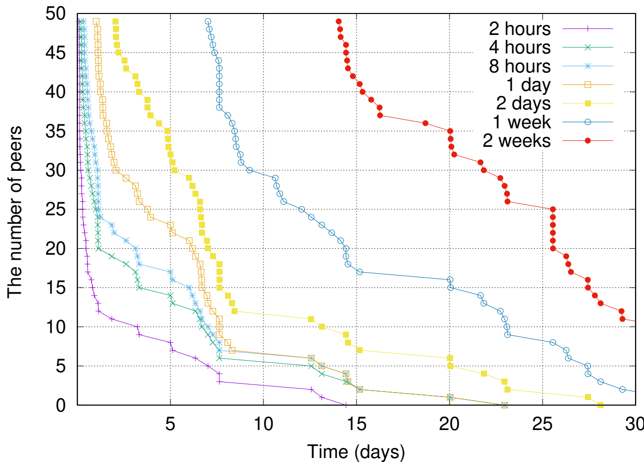


Fig. 16: Connection persistence as a function of client uptime.

F. Timeout values in Geth

Geth defines three hard-coded timeout values in fetching blocks with solicited messages[16]: `arriveTimeout` of 500 milliseconds, `gatherSlack` of 100 milliseconds, `fetchTimeout` of 5 seconds. The time window for batch-processing of block announcements is set to 0.4s (i.e., `arriveTimeout - gatherSlack`). A solicited block is considered failed if unanswered for 5 seconds (i.e., `fetchTimeout`).

As the Gethlighting attack induces mild delay on peer messages in order to expire the above timers at the target (Section IV-B), one may suggest that Gethlighting can be prevented by simply increasing the timeout values (e.g., 5-sec timeout for solicited block request). While an increased message timeout can make Gethlighting attacks a little harder (because attacks need to induce longer delays), the current 5-sec timeout is close to its maximum and it's impractical to further increase it in Ethereum. This is because the timeout is already too close to the average blocktime of 12-sec in Ethereum. An increased timeout (say 10-sec) would force Ethereum nodes to wait so much time for delayed block propagation that they might fail to finish a block reception before the next block is generated.

G. Absence of Ban Score Mechanism in Ethereum

Gethlighting is scalable and low-cost since a low-performance VM (e.g., `t2.large` of Amazon EC2) can comfortably accommodate 10 attack nodes (Section V). During Step-① in Section III, all Gethlighting peers stay idle, except for a periodic ping-pong, meaning that they neither inform any blocks/transactions nor answer to any of the target's requests. As a consequence, Gethlighting nodes' hardware specification does not need to meet the Ethereum Foundation's requirement for blockchain synchorinization.[17]

One may suggest that the cost for Gethlighting can be easily magnified by adopting a peer banning mechanism similar to the Bitcoin's ban score [17]. That is, if the target disconnects misbehaving (e.g., staying idle or not answering to requests) peers by such a mechanism[18], Gethlighting adversary should roll out high-performing, thus more expensive, VMs (e.g., `i3.xlarge`) for blockchain synchronization in order to maintain persistent peer connections to the target. However, it is inherently difficult to distinguish a non-chattering Gethlighting node from a benign bootstrapping node, who is newly synchronizing the chain from the genesis.

[16] See `eth/fetcher/block_fetcher.go`.

[17] https://ethereum.org/en/developers/docs/nodes-and-clients/

[18] Indeed, a candidate mechanism [40] has been designed and implemented by the Ethereum core developers. However, it is abandoned after a heavy discussion on its pros and cons.