NURGLE: Exacerbating Resource Consumption in Blockchain State Storage via MPT Manipulation

Zheyuan He[†¶], Zihao Li[‡¶], Ao Qiao[†], Xiapu Luo[‡✉], Xiaosong Zhang[†✉],
Ting Chen[†✉], Shuwei Song[†], Dijun Liu[§], and Weina Niu[†]
[†]*University of Electronic Science and Technology of China* [‡]*The Hong Kong Polytechnic University* [§]*Ant Group*

***Abstract*—Blockchains, with intricate architectures, encompass various components, e.g., consensus network, smart contracts, decentralized applications, and auxiliary services. While offering numerous advantages, these components expose various attack surfaces, leading to severe threats to blockchains. In this study, we unveil a novel attack surface, i.e., the state storage, in blockchains. The state storage, based on the Merkle Patricia Trie, plays a crucial role in maintaining blockchain state. Besides, we design NURGLE, the first Denial-of-Service attack targeting the state storage. By proliferating intermediate nodes within the state storage, NURGLE forces blockchains to expend additional resources on state maintenance and verification, impairing their performance. We conduct a comprehensive and systematic evaluation of NURGLE, including the factors affecting it, its impact on blockchains, its financial cost, and practically demonstrating the resulting damage to blockchains. The implications of NURGLE extend beyond the performance degradation of blockchains, potentially reducing trust in them and the value of their cryptocurrencies. Additionally, we further discuss three feasible mitigations against NURGLE. At the time of writing, the vulnerability exploited by NURGLE has been confirmed by six mainstream blockchains, and we received thousands of USD bounty from them.**

1. Introduction

Blockchains, with intricate architecture, have evolved various components, e.g., consensus network, smart contracts, decentralized applications, and auxiliary services [1]. These components expose various attack surfaces, leading to severe threats to blockchains [1]. Among these threats, the frequency and severity of Denial-of-Service (DoS) attacks have been rising [2]. DoS attacks deny the service of corresponding components, and compromise the operations of blockchain. For instance, the DoS incident [3], [4] on the consensus network induces a hard fork and abandons over 30 blocks (worth 8.6M USD) atop the Ethereum [5].

Academic communities continuously explore new attack surfaces of blockchain under DoS threats, involving four attack surfaces, i.e., consensus network, txpools, auxiliary services, and smart contracts (cf. details in §8). However, despite serving as the major performance bottleneck of blockchain [6]–[8], DoS security concerning state storage of blockchain has never been explored.

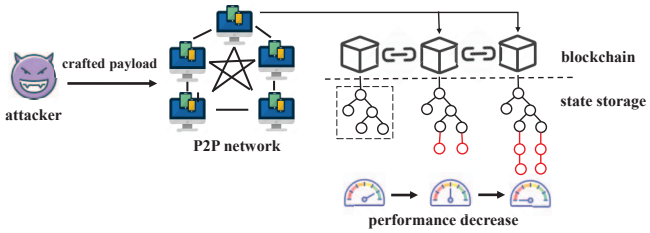


Figure 1: The workflow of NURGLE. The attacker first submits the crafted payload by transactions to the P2P network. Once the transactions are included in the blockchain, they can manipulate the state storage of the MPT structure, and finally impair the performance of the blockchain.

To our best knowledge, we first reveal a new DoS attack surface of blockchain, i.e., state storage, which is used to maintain the blockchain state [8], [9]. Blockchain state is the persistent data of the blockchain (e.g., account balance and contract persistent variables [9], [10]). Besides, blockchain state is managed by a storage structure, typically Merkle Patricia Trie (MPT) [11]. Hence, modifying the blockchain state consumes numerous resources. Specifically, the state storage of MPT structure (termed by *MPT*) utilizes a tree structure to manage blockchain state. A leaf node in *MPT* stores the value of persistent data (e.g., account balance), and all intermediate nodes in the path from the root node to the leaf node correspond to the key of the data's value (e.g., account address) [6]. As the first systematic study on the security of blockchain state storage, our study sheds light on the importance of securing the state storage, and facilitates researchers and developers to propose more robust designs.

Threat goals. In our study, we propose NURGLE, a novel DoS attack towards blockchain state storage. NURGLE aims to cripple blockchain's performance by raising time cost in interacting with its state storage. The design of NURGLE is inspired by our two observations:

- Heavy burden of state maintenance (§4.1). We categorize four classes of time-consuming blockchain operations (Table 1). Besides, we unevil that the time cost of the operations significantly relies on the number of *MPT* nodes involved within the state storage during blockchain execution.
- Flaw of gas mechanism (§4.2). Gas mechanism [12] is responsible for determining the financial cost incurred by

¶ *Co-first authors.*
✉ *Corresponding authors.*

users when accessing blockchain's resources. However, this mechanism fails to consider intermediate nodes in state storage, which are updated when the state is modified (Fig. 5).

According to our two observations, NURGLE can proliferate intermediate nodes with few cost to increase extra resource consumption for maintaining the intermediate nodes, thereby degrading overall blockchain performance. Fig. 1 displays NURGLE's workflow. Specifically, NURGLE first constructs data and submits the crafted payload to the P2P network. Once the payload is included in blockchain, NURGLE can manipulate *MPT*, and finally impair the performance of the blockchain. Furthermore, as the manipulated *MPT* persists on the blockchain, NURGLE persistently threatens the blockchain in all subsequent blocks.

Attack scope. Blockchains, adopting MPT [11] for handling state storage [6], [13], [14], are under threats of NURGLE. In practice, MPT structure is widely applied in mainstream blockchain platforms, e.g., Ethereum [5], Binance Smart Chain (BSC) [15], Polygon [16], Avalanche [17], Optimism [18], and Polkadot [19]. To our best knowledge, 588 blockchain platforms (cf. lists in Appendix C.1) are under threats of NURGLE, which are compatible with Ethereum ecosystem [20], [21]. In addition, other 153 blockchain platforms (cf. lists in Appendix C.1) compatible with Polkadot ecology are also affected by NURGLE [22] (§7).

Attack design. NURGLE strategically manipulates *MPT* to impair blockchain performance. Specifically, NURGLE aims to raise *MPT*'s depth by proliferating intermediate nodes, i.e., nodes between the root node and leaf nodes. When *MPT*'s depth increases, blockchain consumes more resources to maintain expanded intermediate nodes in *MPT*.

Within *MPT*, which is organized as a prefix tree [11], each node is located by a unique value (denoted as *indexing*, e.g., `acd3f` in Fig. 2). Besides, all *MPT* nodes are ordered by the prefixes of their *indexing* [11] (§2). Hence, to proliferate intermediate nodes in a specific path of *MPT*, NURGLE needs to craft a leaf node, whose *indexing* contains a desired prefix (Fig. 4). However, such a task is challenging. This is because, for a leaf node storing blockchain state (e.g., account balance), its *indexing* is derived from the keccak256 hash value [23] of the information used to identify the leaf node (e.g., account address). Hence, to construct a leaf node whose *indexing* contains a specific prefix, NURGLE needs to collide the *indexing* of the leaf node, which is derived by keccak256 hash calculations [23].

Furthermore, in consideration of the trade-off between attack impact and cost, we propose an optimized strategy to reduce NURGLE's cost while retaining most of its original attack's impact (§5.1). We achieve this by selectively deepening the leaf nodes that correspond to the most frequently accessed accounts (i.e., active accounts in §5.1).

Evaluation. To uniformly measure consumed resources of blockchain, e.g., CPU and disk, we utilize the time required for state modification as the metric [24], and comprehensively and systematically evaluate NURGLE in four aspects.

First, we determine reasonable strategies of NURGLE with considering computing resources of launching attacks. Please note that, with commodity hardware resources, NURGLE needs to collide a keccak256 hash value with the specific prefix for manipulating *MPT*. As a result, under the computing resources of single RTX3080 GPU (Table 2), NURGLE can threaten *MPT* by manipulating its structure at the depth of the first 15 layers (§6.1), via colliding the first 13 nibbles of the hash values.

Second, we evaluate NURGLE's impact on Ethereum from the block height of #14.99M to #15M. As a result, with the computing resources of single RTX 3080 GPU, an adversary can persistently increase the time cost of state modification by 111% of Ethereum (§6.2) in a period of 10,000 blocks. We further propose models to estimate attack impact of NURGLE. Our assessment shows that the models can estimate attack impact of NURGLE before launching it.

Third, we further evaluate the financial cost of NURGLE in exploiting seven popular blockchains. Table 3 enumerates the cost of NURGLE. It shows that the lowest cost of NURGLE is 39.64 USD while degrading the performance of Optimism in a period of 10,000 blocks. Besides, our optimization further reduces the cost of NURGLE. Specifically, by striking the active accounts in *MPT* (§5.1), the cost of NURGLE can be further reduced to 3.5% of original cost with retaining 54.66% of the original attack impact. Our results indicate that, by targeting active accounts in *MPT*, the adversary can optimize the cost of NURGLE to a reasonable range (§6.3).

Fourth, we practically evaluate the effectiveness of NURGLE on Ethereum and BSC testnets. Please note that, unlike previous studies [2], [25], [26] which only have non-persistent attack impacts, NURGLE's impact persists in blockchain. Hence, due to ethical concerns, we evaluate the effectiveness of NURGLE in Ethereum and BSC testnets to minimize the potential negative impact. Besides, we carefully adjust attack parameters to light the attack impact. As a result, when we witnessed that NURGLE caused the time of state modification on Ethereum (resp. BSC) testnet to increase by 15% (resp. 18%), we ceased the attack (§6.4).

At the time of writing, vulnerabilities under NURGLE have been confirmed by six blockchains (i.e., Ethereum, BSC, Polygon, Optimism, Avalanche, and Ethereum Classic), and we received thousands of USD bounty from them.

Contributions of this work are listed as follows

- *Novel attack at new attack surface.* Based on a new attack surface (i.e., the state storage of blockchain), we propose a novel DoS attack, NURGLE. By manipulating the MPT structure of state storage, NURGLE can persistently aggravate the consumed resources during blockchain state modification, including CPU, memory, and disk resources.
- *New observations.* To our best knowledge, we are the first to categorize the four classes of heavy time-consuming operations of blockchain. Besides, we reveal the flaw of gas mechanism, i.e., it fails to accurately reflect the actual consumed resources of state modification in *MPT*. Our two observations further inspire the design and the mitigation strategies of NURGLE.
- *New understandings.* We conduct a comprehensive and systematic evaluation of NURGLE, including assessing factors affecting NURGLE, evaluating the attack impact

of NURGLE, measuring financial cost of NURGLE in exploiting seven mainstream blockchains, and validating the effectiveness of NURGLE on two testnets. Our experimental results demonstrate that NURGLE can widely exploit various mainstream blockchain platforms, leading to a significant degradation of blockchain's performance at a reasonable cost. Furthermore, we discuss the severe implications brought by NURGLE.

- *Mitigations.* We elaborate on three classes of feasible mitigations that can reduce the attack impact of NURGLE, and discuss their advantages and disadvantages.

We release materials of our work in https://github.com/hzysvilla/Nurgle_Oakland24 for future research.

2. Background

We introduce basic concepts of blockchain (§2.1), explain how blockchain adopts the MPT structure to maintain its state storage (§2.2), and provide an example to illustrate how to exacerbate consumed resource in *MPT* (§2.3).

2.1. Blockchain basic concepts

We use the implementation of Ethereum [5] to introduce the basic knowledge of blockchain, because Ethereum is a widely used blockchain platform. The native cryptocurrency of Ethereum is Ether. According to its specification [5], the basic structure of its data is the block, which comprises the block header and block body. The block header involves a reference to the preceding block and the information used for state validation [5], while the block body contains a sequence of transactions. The transactions are signed by users to transfer funds and communicate with smart contracts.

There are two types of accounts in Ethereum, i.e., smart contract accounts (CA) and externally owned accounts (EOA). An EOA is controlled by a private key held by a user, while a CA holds the pre-defined logic and persistent variables. A contract is a Turing-complete automation program on Ethereum. The execution of contracts is facilitated by the Ethereum Virtual Machine (EVM), which is an underlying component of Ethereum supporting a set of instructions [5].

The gas mechanism [12] establishes the cost associated with users utilizing the blockchain's resources. For example, each operation in Ethereum, which modifies the state data, will introduce the cost of gas, e.g., executing contracts and transferring funds (e.g., Ether) [5].

2.2. Blockchain state storage of MPT structure

In this study, we focus on blockchains like Ethereum, whose state storage is organized in MPT structure (i.e., *MPT*), comprising account data and contract data [8].

In Fig. 2, we display the state storage of MPT structure (left), and its flat layout (right). Account Data 1 maintains all accounts' information, e.g., balance, nonce, code hash, and storage root of each account. Besides, the account information of each account is separately reserved in a unique

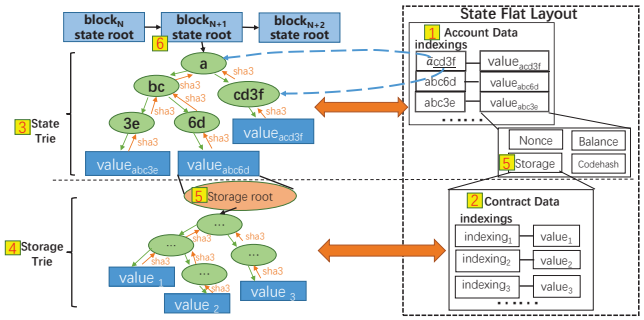


Figure 2: The left part shows state storage in the MPT structure. We display the flat layout of corresponding information at the right part to facilitate intuitive understanding.

leaf node. In Account Data 1, the account information of each account is mapped by the *indexing* of the leaf node reserving the account information. As mentioned in §1, the *indexing* (e.g., `abc6d`) is used to locate a leaf node in the MPT structure [11]. If an account is a contract, its storage root 5 in Account Data 1 points to its Contract Data 2. Contract Data 2 contains the data of the contract to store persistently [5]. Specifically, Contract Data 2 is a mapping from the slots [27] of a contract's storage to the values stored in corresponding slots [28]. In MPT structure, each data, mapping to a contract storage slot, is reserved in a unique leaf node, and the *indexing* of the leaf node is derived from its slot. In addition, each contract has an independent Contract Data 2 [5]. Account Data 1 and Contract Data 2 are maintained in State Trie 3 and Storage Tries 4, respectively. Besides, both State Trie 3 and Storage Tries 4 are in MPT structure for managing and verifying state.

MPT structure manages and indexes data by compressing the prefix tree [29]. In *MPT*, the *indexing* of leaf nodes is a 256-bit byte array [5], i.e., containing 64 nibbles. Please note that, we denote the length of *indexing* as the number of nibbles (i.e., half byte) in it. Taking the *indexing* `abc6d` (Fig. 2) as an instance, the length of its *indexing* `abc6d` is 5. To obtain the data reserved in the leaf node, we need to locate the leaf node with the *indexing* `abc6d` by searching for the intermediate nodes holding the prefix of `abc6d` (i.e., the three intermediate nodes `a`, `bc`, and `6d` in Fig. 2).

MPT performs the state verification for consensus in the idea of the Merkle tree [30]. Each parent node maintains the keccak256 hash value of all its child nodes' data. The keccak256 hash value of the root node atop State Trie 3 is used as the metadata of a block's header, named as state root 6. The orange arrow in Fig. 2 displays the procedure of verifying state. When the state of an account (e.g., Ether balance) changes, since the account's data is stored on a leaf node, it will force all the hash values from the leaf node to the root node to be changed, and the information within nodes in the path is required to be updated accordingly. Taking the account with the *indexing* `acd3f` in Fig. 2 as an example, when the information of the account changes, the hash values maintained by nodes `a` and `cd3f` will be changed, resulting in both the two nodes to be updated.

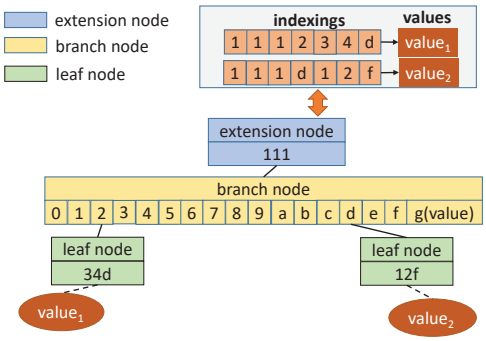


Figure 3: A simplified *MPT*, constructed by two leaf nodes with *indexing* as `111234d` and `111d12f`. The extension node (`111`) handles the common prefix (`111`) of the two leaf nodes' *indexing*. The branch node forks to point the two leaf nodes by two pointers (i.e., `2` and `d`). Finally, the two leaf nodes keep the unique part of their *indexing* as `34d` and `12f`.

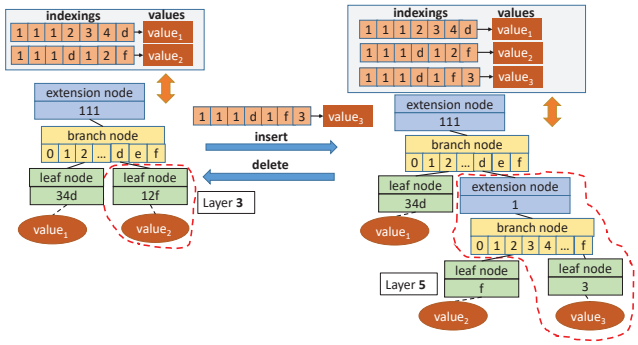


Figure 4: After inserting a leaf node `111d1f3` into the *MPT*, the leaf node `12f` splits into an extension node, a branch node, and two leaf nodes. Finally, the *MPT* is proliferated with new nodes, and its depth is deepened into two layers. Newly generated nodes are marked with red dotted lines.

2.3. Exacerbating consumed resource in *MPT*

We depict the detailed design of *MPT* in Ethereum, and illustrate an example of MPT manipulation, which causes the resource consumption of Ethereum to be exacerbated. There are three types of nodes in Ethereum's *MPT*, i.e., branch nodes, extension nodes, and leaf nodes. A branch node stores up to 16 pointers (from pointer `0` to pointer `f`). A pointer can point to a leaf node, an extension node, or another branch node. An extension node reserves a byte sequence, which is used to point to nodes with the common hash sequence. An extension node also contains a pointer to its child node. A leaf node stores a pointer targeting the data reserved in the leaf node (e.g., Ether balance). We deliver a simplified *MPT* in Fig. 3, where there are two leaf nodes storing data. The *indexing* of the two leaf nodes are `111234d` and `111d12f`. The common prefix of the two *indexing* is `111`, which is handled by the extension node `111`. The branch node forks to point the two leaf nodes by utilizing the two pointers `2` and `d`, respectively. Finally, the two leaf nodes keep the unique part of their *indexing*, i.e., `34d` and `12f`.

Inserting and deleting nodes can trigger the node splitting and collapse in *MPT*, which leads to extra resource consumption [31]. Fig. 4 illustrates the node splitting triggered by inserting a leaf node (whose *indexing* is `111d1f3`) into the *MPT*. Since the *indexing* `111d1f3` of the inserted leaf node has a common prefix (i.e., `111d1`) with the *indexing* `111d12f` of an existing leaf node in the *MPT*. Therefore, the leaf node (i.e., `12f`) is split into an extension node `1`, a branch node (containing two pointers as `2` and `f`), and two leaf nodes (i.e., `f` and `3`). During the whole procedure of inserting the leaf node `111d1f3`, it additionally consumes resources (e.g., CPU, memory, and disk resources) for maintaining and verifying the newly generated intermediate nodes. Furthermore, the node collapse will happen by deleting leaf nodes from *MPT*. Taking the right part of Fig. 4 as an example, during deleting the leaf node `111d1f3` from the *MPT*, it additionally consumes resources to discard nodes (which are marked with red dotted lines), and re-verify all involved nodes.

3. Threat model

The threat model of NURGLE involves two actors: an adversary and a victim blockchain. The adversary submits crafted payload by transactions to the victim blockchain, resulting in impairing the blockchain's performance. In the threat model, the victim blockchain supports the execution of smart contracts, and adopts the MPT structure to maintain and update its state storage. Besides, there is a P2P network atop the victim blockchain, accepting users to submit transactions to be included in the victim blockchain.

The adversary controls an externally owned account (EOA) and necessary assets for sending transactions to the P2P network of the victim blockchain. Besides, the adversary controls a modified client of the victim blockchain, by which the adversary can monitor and analyze the status quo of the victim blockchain's *MPT* to construct the attack payload. Moreover, the adversary has limited resources, e.g., GPUs, for constructing the payload to mount attack.

It is reasonable for a financially rational adversary to launch NURGLE for two reasons. i) NURGLE can yield economic returns by arbitraging during the price volatility of cryptocurrency assets on the victim blockchain [25]. For example, the adversary can first utilize NURGLE to impair the performance of the victim blockchain and weaken the trust of related ecology, leading to a decline in the asset price of the cryptocurrency. After that, the adversary can just conduct the short selling [32] on the corresponding cryptocurrency assets to obtain a considerable profit. ii) Adversaries running the blockchain can utilize NURGLE to disrupt competitors. By striking competitors, it drives the customer base of the victim, flocking to the adversaries [25].

The cost of NURGLE consists of two parts, i.e., the fee of computing resources and the gas fee. The major fee of computing resources comes from GPUs, which are used to collide a leaf node with a desired prefix for inserting it into a target position of the *MPT* (cf. details in §5.1). It is necessary to undertake gas fee for the adversary because the adversary needs to submit transactions with crafted payload in the victim blockchain for launching NURGLE (§5.1).

TABLE 1: Four operations that involve heavy burden of state maintenance

Operation type	Description	Major consumed resources
[OP1] MPT update	During the state modification, the *MPT* will update nodes in *MPT* (i.e., nodes in State Trie 3, and Storage Tries 4) that are required to be modified.	Disk read, and memory read and write
[OP2] MPT verification	To verify whether *MPT* holds the latest state, *MPT* derives the keccak256 hash value of the root node of the whole State Trie 3 from all other nodes in *MPT* at a bottom-up manner.	CPU computation
[OP3] MPT holding in memory	To mitigate consumed disk resource in accessing nodes in *MPT*, the blockchain holds partial nodes in *MPT* (e.g., nodes in State Trie 3 and Storage Tries 4) in memory, and can determine which nodes holding in memory to be discarded.	CPU computation, and memory read and write
[OP4] MPT persistence	*MPT* persistently stores nodes representing the latest state of blockchain into disk.	Disk write

4. Observation of blockchain state storage

In this section, we illustrate two observations, i.e., the heavy burden of state maintenance (§4.1) and the flaw of gas mechanism (§4.2), which inspires the design of NURGLE.

4.1. Heavy burden of state maintenance

Operations with *MPT* (e.g., maintaining, verifying, modifying, and assessing state in *MPT*) are the major performance bottleneck of blockchain. Previous studies [6]–[8] report that over 81% execution time of blockchain costs in interacting with *MPT*. To make it worse, we further uncover that the time cost of major time-consuming operations in interacting with *MPT* linearly increases with the number of nodes in *MPT* involved in these operations, leading the performance of blockchain heavily depend on the number of nodes in *MPT*. We categorize the operations into four classes in Table 1 and depict them as follows.

- **OP1 (MPT update).** Blockchain updates nodes in *MPT* involved in state modification (e.g., node splitting in §2.3). In **OP1**, both nodes, corresponding to Account Data 1 in State Trie 3 and Contract Data 2 in Storage Tries 4, can be modified. For example, when an account's balance changes, the leaf node (corresponding to the account) in State Trie 3 will update its reserved data, e.g., updating to the account's latest balance. **OP1** consumes disk read, and memory write and read, because intermediate nodes and leaf nodes associated with the account can be absent in current memory, and they have to be accessed from the disk accordingly in such cases.
- **OP2 (MPT verification).** According to §2.2, to achieve the consensus of blockchain state, *MPT* needs to derive the keccak256 hash value of the root node of the whole State Trie 3 from all other nodes in *MPT* at a bottom-up manner (§2.2). We denote the operation for computing the hash value of the root node as MPT verification (**OP2**), since the hash value is used to verify whether *MPT* holds the latest state during consensus. Specifically, **OP2** will first derive the hash value of the root of each Storage Trie 4 from all nodes in the Storage Trie 4, and then calculate the hash value of the root node of the whole State Trie 3 (§2.2). **OP2** consumes expensive CPU resource to calculate the keccak256 hash value for each node [7].
- **OP3 (MPT holding in memory).** To ease consumed disk resource in accessing nodes in *MPT*, blockchain holds partial nodes of *MPT* (e.g., nodes in State Trie 3 and Storage Tries 4) in memory, and can determine which nodes holding in memory to be discarded from memory. We denote the operation for holding nodes of *MPT* in memory as **OP3**. Besides, discarded nodes are expired nodes that have been replaced by other latest nodes during state changes [33]. Since discarded nodes will not be written to disk, **OP3** mainly consumes CPU and memory resources for maintaining nodes of *MPT* in memory.
- **OP4 (MPT persistence).** The *MPT* persistently stores nodes representing the latest state of blockchain into disk (**OP4**), and generates heavy overhead of disk writes.

Please note that, **OP1-4** can trigger each other. For Fig. 4 as an example, once the intermediate nodes and leaf nodes are modified in **OP1**, the keccak256 hash value of them should be updated, including the root node of the whole State Trie 3 (**OP2**). Besides, since new nodes are generated in *MPT*, the nodes of *MPT* in memory can also be replaced and discarded (**OP3**). Furthermore, the newly generated nodes can also be written into disk for persistently maintaining (**OP4**). In addition, the time complexity to finalize all **OP1-4** is $\mathcal{O}(n)$, where n is the number of nodes in *MPT* involved in **OP1-4**. It means that the more MPT nodes need to be maintained, the more resources will be consumed. Our first observation assists NURGLE to exacerbate consumed resources by involving more nodes in *MPT* to increase more time cost for handling the four classes of operations.

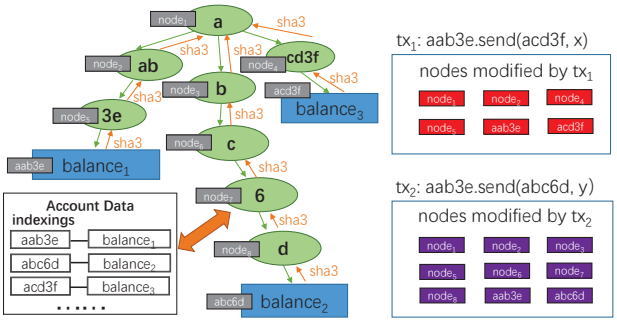
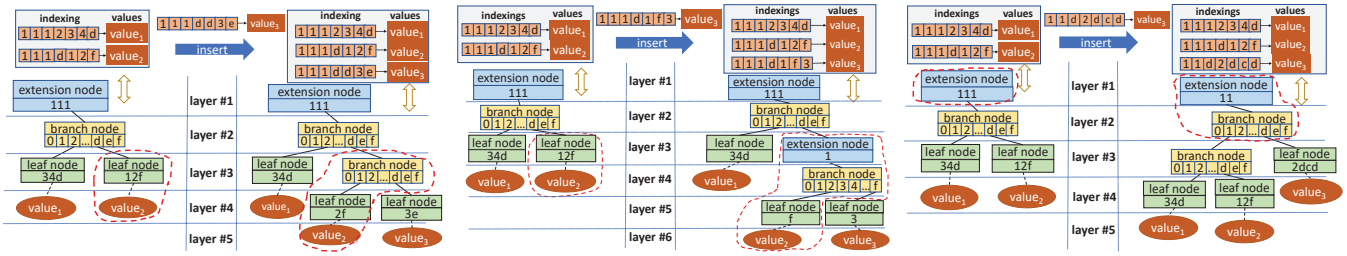


Figure 5: There are two transactions tx_1 and tx_2 transferring Ether, and each of them consumes 21,000 units of gas. However, when executing tx_1 and tx_2, the number of nodes in *MPT* to be updated (§2.2) is different (marked as red and purple boxes in Fig. 5). After finalizing the two transactions, although the two transactions cost the same amount of gas, they consume different resources for updating the *MPT*.



(a) An example of Strategy 1 (b) An example of Strategy 2 (c) An example of Strategy 3

Figure 6: Examples of utilizing the three strategies (**S1-3**). Nodes before and after splitting are marked with red dotted lines.

4.2. Flaw of the gas mechanism

To our best knowledge, we are the first to uncover the flaws of gas mechanism in the view of state storage. We use an example in Fig. 5 to illustrate the flaw of gas mechanism revealed by us. There are three EOAs in Fig. 5. The leaf nodes reserving the three EOAs' information are located by their *indexing* as `aab3e`, `abc6d`, and `acd3f` (marked as blue boxes). In the example, there are two transactions (i.e., tx_1 and tx_2), where tx_1 transfers Ether from `aab3e` to `acd3f`, and tx_2 transfers Ether from `aab3e` to `abc6d`. According to Ethereum's specifications [5] , each of tx_1 and tx_2 costs 21,000 units of gas. However, the resources consumed by the two transactions are different. Specifically, during the execution of tx_1, six nodes (marked as red boxes) in the *MPT* are required to be updated. However, during the execution of tx_2, nine nodes (marked as purple boxes) are required to be updated. Therefore, although the two transactions cost the same amount of gas, the amount of resources consumed by them is different for updating the *MPT*. Similarly, the flaw of gas mechanism can also be observed in updating the data reserved in contract storage (Storage Tries 4). The root cause for the flaw is that the current gas mechanism (e.g., gas mechanism of Ethereum [5]) does not consider the exact resources consumed for maintaining and verifying state in *MPT* (e.g., the resources for modifying intermediate nodes). Our second observation inspires NURGLE to exacerbate consumed resources of state modification by introducing more intermediate nodes involved in the state modification.

5. The design and implementation of NURGLE

In this section, we elaborate on the design of NURGLE (§5.1) and NURGLE's detailed implementation (§5.2).

5.1. The design of NURGLE

Inspired by our two observations (§4), NURGLE aims to expand intermediate nodes of *MPT* by inserting leaf nodes into desired positions of *MPT*. Specifically, to expand the intermediate nodes in *MPT*, we utilize the node splitting (§2.3) triggered while inserting leaf nodes into *MPT*. After that, the expanded intermediate nodes will increase the consumed resources for the operations of modifying, updating, and verifying nodes in *MPT* (e.g., **OP1-4** in §4.1).

Inserting a leaf node in *MPT* by different strategies can trigger node splitting in different ways, causing distinct results (§2.3). In the following, we categorize three strategies, which split nodes and deepen a target leaf node ($node_v$) by inserting a leaf node ($node_{insert}$) in different ways. We denote that, the length of the whole *indexing* of $node_v$ is m, and the length of the unique part of $node_v$ is n. For the leaf node `acd3f` in Fig. 2 as an example, the length of its whole *indexing* and the length of its unique part are 5 and 4, respectively.

- **S1**. $node_v$ splits into a leaf node, and a branch node, when the length of the common prefix between the *indexing* of $node_v$ and $node_{insert}$ equals to m-n. **S1** deepens $node_v$ by 1 layer, and adds an intermediate node. For example, when inserting the leaf node `111dd3e` (Fig. 6a), leaf node (whose unique part is `12f`) splits into a branch node (containing the two pointers `1` and `d`), and a leaf node (whose unique part is `2f`).
- **S2**. $node_v$ splits into an extension node, a branch node, and a new leaf node, when the length of the common prefix between the *indexing* of $node_v$ and $node_{insert}$ is larger than m-n. **S2** deepens $node_v$ by 2 layers, and adds two intermediate nodes. For example, when inserting the leaf node `111d1f3` (Fig. 6b), the leaf node (whose unique part is `12f`) splits into an extension node `1`, a branch node, and a leaf node (whose unique part is `f`).
- **S3**. An extension node in the path from the root node to $node_v$ splits into a branch node and a new extension node,

Algorithm 1: NURGLE

Input: $node_V$, the leaf node to be deepened in *MPT*
Input: $node_{root}$, the root node of *MPT*
Output: $nodes_{insert}$, the leaf nodes to be inserted for deepening $node_V$

```
nodes_insert ← []
nodes_collided = HashCollision(node_V)
do
    nodes_intermediate = TraverseNodes(node_root, node_V)
    for node ∈ nodes_intermediate do
        if Type(node) is "extension" then
            if IsSplittable(node) then
                node_insert = MatchS3(node, nodes_collided)
                if node_insert ≠ null then
                    nodes_insert.append(node_insert)
        else if Type(node) is "leaf" then
            if IsSplittable(node) then
                node_insert = S2Match(node, nodes_collided)
                if node_insert ≠ null then
                    nodes_insert.append(node_insert)
                else
                    node_insert = S1Match(node, nodes_collided)
                    if node_insert ≠ null then
                        nodes_insert.append(node_insert)
while New node_insert has appended into nodes_insert;
return nodes_insert
```

when i) the common prefix between the *indexing* of $node_v$ and $node_{insert}$ cannot cover the prefix maintained in the extension node, and ii) the length of the prefix maintained in the extension node is larger than 1. **S3** deepens $node_v$ by 1 layer, and adds an intermediate node. For example, when inserting leaf node `11d2dcd` (Fig. 6c), extension node (maintaining `111`) splits into an extension node `11` and a branch node.

Algorithm 1 presents the process of NURGLE. NURGLE crafts a list of leaf nodes (i.e., $nodes_{insert}$). By inserting in *MPT*, $nodes_{insert}$ expand intermediate nodes by node splitting and deepen $node_v$. Specifically, for a given leaf node $node_v$, under a predefined timeout, NURGLE collides the *indexing* of $node_v$ with aiming to maximize the length of common prefix between $node_v$ and collided nodes (Line 2). After the timeout, NURGLE collects all collided nodes in $node_{collided}$. In Line 4, NURGLE then retrieves all nodes in the path of *MPT* from the root node of *MPT* to $node_v$ (i.e., $nodes_{intermediate}$). NURGLE iterates nodes in $nodes_{intermediate}$ to determine whether they can be split by node splitting (Line 7 and 12). If a node in $nodes_{intermediate}$ can be split, NURGLE generates the leaf node $node_{insert}$ for splitting the node, by matching whether the three strategies are satisfied (Line 8, 13, and 17). NURGLE then collects $node_{insert}$ into $nodes_{insert}$ (Line 10, 15 and 19). NURGLE will continue the whole procedure (Line 4 - Line 19), until there are no more new leaf nodes for node splitting. It is worth noting that, during the whole procedure (Algorithm 1), to improve the efficiency of NURGLE, we also record the nodes that can not be split, which helps NURGLE to directly skip these nodes in Line 7 and 12.

For a leaf node in *MPT*, Lemma 1 guarantees that NURGLE can deepen the leaf node by triggering leaf splitting.

Lemma 1. *Given a leaf node* $node_v$ *that the length of the unique part of its* indexing *is greater than 2, if* NURGLE *can collide out nodes* $nodes_{insert}$ *whose common prefix length with* $node_v$ *is at most* x*, then* $node_v$ *can be deepened up to the layer* $x+2$ *(where* $x+2$ *is no larger than the maximum depth of* MPT*) by inserting* $nodes_{insert}$ *in* MPT. □

Proof of Lemma 1. Under node splitting triggered by NURGLE, the first x nibbles of *indexing* of $node_v$ will be maintained in x intermediate nodes, because leaf nodes crafted by NURGLE can deepen $node_v$ (**S1-2**) and split the extension nodes whose maintained prefix length is larger than 1 (**S3**), and the length of pointers maintained in a branch node is 1 (§2.3). Hence, $node_v$ locates in the layer $x+1$ in *MPT* (i.e., x intermediate nodes are in front of $node_v$). NURGLE can then deepen $node_v$ by 1 layer (i.e, deepening to the layer $x+2$), with the length of the common prefix between the *indexing* of $node_v$ and the inserted leaf node equaling to x (**S1**). □

To trigger node splitting for a leaf node, NURGLE needs to craft leaf nodes, whose *indexing* has a common prefix with the leaf node, and insert crafted leaf nodes in *MPT* (**S1-3**). However, it is challenging, because *indexing* of the leaf node is derived from keccak256 hash computation, and hash algorithm is irreversibility [34] (§1). To address the challenge, we design new methods to craft leaf nodes triggering node splitting by colliding the prefix of target leaf node's *indexing*. Specifically, based on the parallel computing of GPUs, we adopt OpenCL library [35], [36] to collide the *indexing* by parallelized computing keccak256 hash (Appendix A.1).

Multi-target hash collision. There are multiple leaf nodes in *MPT*, whose *indexing* is required to be collided by NURGLE. A trivial idea is to collide each leaf node's *indexing* one by one. We denote the counts of hash calculations to collide a specific *indexing* as θ. Hence, the expected number of keccak256 hash calculations required for hash collision (denoted as E_ϕ) grows linearly with the number of leaf nodes whose *indexing* is required to be collided (denoted as ϕ), i.e., by the trivial method, $E_\phi = \theta \times \phi$. Compared with the trivial idea, we propose a new multi-target hash collision strategy to collide all target leaf nodes' *indexing* simultaneously, and by which, E_ϕ decreases to be as $\theta \times ln(\phi)$.

Lemma 2. *Given* ϕ *leaf nodes whose* indexing *is required to be collided, if* NURGLE *costs* θ *keccak256 hash calculations to collide a specific* indexing*,* NURGLE *costs* $\theta \times ln(\phi)$ *keccak256 hash calculation to collide all target* indexing *simultaneously by multi-target hash collision.* □

Proof of Lemma 2. According to the multi-target collision search [37], [38] which investigates how many calculation counts are required to achieve one collision against multiple targets, NURGLE costs $\frac{\theta}{\phi}$ calculations to collide a *indexing* of all target leaf nodes. In addition, according to the coupon collector's problem [39], E_ϕ grows with ϕ as the complexity of $\mathcal{O}(n \times ln(n))$ [39]. Hence, under our multi-target hash collision strategy, the required calculation counts for achieving hash collision against all target leaf nodes (E_ϕ) equals to the product of the calculation counts for colliding a *indexing* of all target leaf nodes ($\frac{\theta}{\phi}$) and the complexity of how E_ϕ grows ($\phi \times ln(\phi)$). Hence, E_ϕ can be derived by Eq. 1. □

$$E_\phi = \frac{\theta}{\phi} \times \phi \times ln(\phi) \tag{1}$$

Leaf node insertion. After obtaining the leaf nodes to be inserted in *MPT*, NURGLE uses different methods to insert leaf nodes in State Trie 3 and Storage Tries 4, respectively, e.g., transferring 1 wei Ether (i.e., 10^{-18} Ether) to a target EOA account. We elaborate on the methods in the following.

• Insert leaf nodes in State Trie 3. The *indexing* of a leaf node in State Trie 3 is derived from the address of an EOA. Hence, after the hash colliding, NURGLE will finally determine an EOA in such cases. To insert the corresponding leaf node in *MPT*, NURGLE directly sends 1 wei Ether (i.e., 10^{-18} Ether) to the EOA account by transactions.

• Insert leaf nodes in Storage Tries 4. Storage Tries 4 hold the persistent data for a contract's storage, and the *indexing* of a leaf node in Storage Tries 4 is derived from the slot of the contract storage. To insert a leaf node in Storage Tries 4, NURGLE can only modify the data reserved in corresponding slot by interacting with the contract [5]. Please note that, for a key-value pair in a mapping [40], e.g., `k` and `v`, `v` stores in a storage slot, and the slot is derived from `k` by keccak256 hash computation. Hence, to insert a leaf node corresponding to a specific slot, NURGLE crafts elements (e.g. `k`) in mappings of contracts. Specifically, after hash

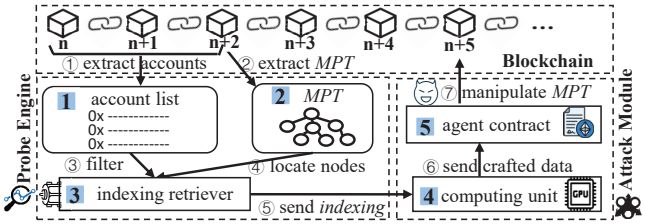


Figure 7: The implementation consists of seven steps. Step ① and ②: NURGLE extracts the list of all accounts 1 and *MPT* 2 from blockchain. Step ③: The indexing retriever 3 retrieves target accounts from the list of all accounts 1. Step ④: The indexing retriever 3 locates target leaf nodes in *MPT* 2, and derives *indexing* of them. Step ⑤: The indexing retriever 3 forwards derived *indexing* to the computing unit 4 for colliding *indexing*. Step ⑥: The computing unit 4 generates the crafted data (e.g., addresses of accounts) from *indexing* with the desired prefix, and sends the crafted data to agent contract 5. Step ⑦: The agent contract 5 manipulates *MPT* by inserting crafted leaf nodes.

colliding, NURGLE determines the parameters [41], [42] to invoke a function in the target contract, which will update data in the target slot (corresponding to the target leaf node in Storage Tries 4). Taking ERC20 token contract [43]–[45] as an example, NURGLE finally determines the parameters for invoking `transfer()` by transferring the smallest unit of token (e.g., 10^{-6} USDT) to a target account address, where `transfer()` inserts the crafted leaf node in Storage Tries 4, and set the data of the desired slot to be 1.

Optimization strategy. To launch NURGLE, its cost should be considered (§3). Expanding intermediate nodes associated with all accounts is impractical, because there are billions of leaf nodes in *MPT*, and the corresponding cost is beyond the limited computing resources and assets of the adversaries in our model (§3). To reduce the cost, we propose an optimized strategy of NURGLE to only deepen the leaf nodes associated with active accounts. Active accounts are the accounts that keep conducting frequent trades over a period of time [8], and they can be trivially captured by querying the frequency of each account being accessed and modified from blockchain. According to the captured list, adversaries can strategically delineate the range of leaf nodes deepened by NURGLE, e.g., the leaf nodes associated with the accounts modified no less than six times in a specific range of time. Since nodes associated with active accounts are keeping updating the reserved data, the resource consumption brought by NURGLE will be further exacerbated with limited cost.

5.2. The implementation of NURGLE

Fig. 7 shows the overview of NURGLE's implementation. There are seven steps in NURGLE encompassing three core portions: i) Blockchain serves as the data source and attack target. ii) Probe engine analyzes the accounts information and *MPT*, and retrieves the *indexing* of target leaf nodes. iii) Attack module wields computing resources to generate the crafted data from the retrieved *indexing*, and invokes the agent contract to insert the crafted leaf nodes into *MPT*. We portray the implementation of NURGLE by steps ① to ⑦.

Blockchain. In Step ① and ②, NURGLE collects the list of all accounts 1 and *MPT* 2 from blockchain. The account list 1 is used to capture active accounts. Besides, we record the frequency of each account being accessed and modified in the account list 1. *MPT* 2 is used to retrieve the *indexing* of active accounts' leaf nodes for proliferating intermediate nodes. We instrument logic of ① and ② in blockchain client, and run the client in real-time for data collection.

Probe engine. Probe engine retrieves *indexing* of target leaf nodes by analyzing the list of accounts 1 and *MPT* 2, and NURGLE aims to trigger node splitting and proliferate intermediate nodes associated with the leaf nodes. In Step ③, Probe engine first retrieves active accounts from the account list 1. We determine an active account when the frequency of it being modified and accessed is larger than a threshold (e.g., 6) in a specific block range [8]. In Step ④, the indexing retriever 3 retrieves the leaf nodes and their *indexing* corresponding to active addresses from *MPT* (§5.1). In step ⑤, the indexing retriever 3 sends obtained *indexing* to the Attack module for the computing of hash collision.

Attack module. The Attack module generates the corresponding crafted data through the computing unit 4, and then inserts the crafted leaf nodes into *MPT* through the agent contract 5. Computing unit 4 utilizes GPU resources for hash computing. The agent contract 5 is deployed by the adversary, and Fig 8 displays the code snippet of the agent contract. In step ⑥, NURGLE leverages the computing unit 4 to collide *indexing* with desired prefix to generate the crafted data (e.g., the address of an account). Computing unit 4 then sends the crafted data to the agent contract 5.

```
1  contract NurglePrototype{
2    function NurgleState(address payable[] memory
         Payloads) payable public{
3      uint256 len = Payloads.length;
4      for(uint256 i=0; i < len; i++){
5        bool result=Payloads[i].send(1);}}
6    function NurgleStorage(address dst, bytes4 funcsig,
         bytes[] memory Payloads, uint num) public {
7      uint256 len = Payloads.length/num;
8      for(uint256 i=0; i < len; i++){
9        bytes memory encode=abi.encodePacked(funcsig);
10       for(uint256 j=num*i;j <num*i+num;j++){
11         encode=abi.encodePacked(encode, Payloads[j]);}
12       dst.call(encode);}}}
```

Figure 8: Code snippet of agent contract. `NurgleState()` (Line 2-5) inserts the crafted leaf nodes into State Trie 3 by sending 1 wei Ether to target accounts. `NurgleStorage()` (Line 6-12) inserts crafted leaf nodes on Storage Tries 5. As the logic of how smart contracts access their Storage Tries 5 can be distinct, `NurgleStorage()` allows the adversary to customize: i) `dst`, the callee contact, ii) `funcsig`, the function to be invoked, and iii) `Payloads`, the parameters for invoking the function. In Line 9-11, `NurgleStorage()` splices `funcsig` and `Payloads`. In Line 15, `NurgleStorage()` invokes `dst` with crafted payload, which executes the logic of `dst` to insert the leaf nodes on Storage Tries 5.

TABLE 2: Time cost for NURGLE to collide different lengths of desired prefix for an *indexing*.

Digits	Crafted data (hex encoding)	*indexing* (hex encoding)	Time cost
1	0x51b0e4b84afc9c7e935fd1c54409abda46ffff07	0x109999afd60b733da226a060260c2d9f165f0f33516c5a3230d2b9538ae197e7	*<1s*
2	0x7c0caee5b72d0c71a090c6f02522e89acfffff07	0x11fb9e6a64c5a7c23fb27d08e3d74ea1018fcb0c60d2010cca6c6654dd95e4b8	*<1s*
3	0x8f5ea3c9db43de4143e5717f44dcb43e05d0fe07	0x1110dc62b86ce4609e860381909da5480d46b2e90ea19c5afac287be805c234b	*<1s*
4	0xbd6f8cba28b4a0218d0aedbc820a27248ee4fe07	0x111165e10752633a1ab85c219c618d6c6e6259fdb7c8d2397df9cb72d16e4149	*<1s*
5	0xfccedcfd14858e8b1baf9a497e99af468012b507	0x111110e0c5d11a713c428c42a03a5a7c55d66c0e61158ef13a63776b94d384d0	*<1s*
6	0x58b91f9cb0ffacae5d95c9e80c373d264993cc06	0x111111078c719cdc5abc2195b645a72ba7dd4d15b12ab9cce3361466c402df69	*<1s*
7	0x89f25e63c12c48a95c22cd4b19585f337a805f06	0x1111111106b6090ca5f7027a7539dc73173e26a35b28645b47d4878db6bbddd62	*<1s*
8	0xa0f0722109f07edd76cc1d2b29cfbc0122ca2b06	0x11111111ce35790ede4c97cc847e55c91c0b3063f5cb56ab6ab93ee76381fa6a	*<1s*
9	0x97637e992f835689667a48a0731ce1ebb44dc006	0x1111111110b3bf4ed6dc409fb20328970a0f23dac93761a4347fcd4c84dfe8cc	*53s*
10	0x2f1033b78f8fb3c04259202793d2d89169326d02	0x1111111111ce8bad4529bfef324c88454fe4e72c3cd3974c0249c9adc764802a	*21.68m*
11	0x267a239f1986295e996358a79f57b473ae264d05	0x111111111100822f67e0319be36eb814ade0ca60c65c62b41641e889eb48ad8	*2.8h*
12	0xd4dfd776a81fcdfa2d601f1efa31a2ad8c21fe06	0x111111111111834eea3006374356f398b29f9b709272533e759348f0bb07aa11	*12.57h*
13	0xdf04b72b67666a59ff30c06dd079f1850b36ba04	0x1111111111111ca536d3de683a3ab986f631ee733132457eccc0d9a011aa9e55	*24.58h*

In Step ⑦: The agent contracts 5 insert crafted leaf nodes into *MPT* by invoking its functions (i.e., `NurgleState()` and `NurgleStorage()`). `NurgleState()` (Line 2-5) inserts crafted leaf nodes in State Trie 3 by sending 1 wei Ether to target accounts. `NurgleStorage()` (Line 6-12) inserts crafted leaf nodes in Storage Tries 5 by invoking specific functions of target contracts (e.g., `transfer()` function of ERC20 token contract) with crafted payload.

6. Evaluation

We answer four research questions for evaluating NURGLE's cost and impact. **RQ1:** How do computing resources influence the attack impact of NURGLE? **RQ2:** How severe is the attack impact of NURGLE on the current blockchain? **RQ3:** How is the economic feasibility of NURGLE? **RQ4**: Will NURGLE threaten the current blockchain in practice?

Experimental setup. We evaluate NURGLE on a server with an Intel Xeon Gold 5218R CPU (2.10 GHz, 10 cores), 64 GB RAM, 1 TB SSD, and single RTX3080 GPU. We adopt a go-ethereum client at v1.11.6 [46] to measure the consumed resources of blockchain. We evaluate the impact of NURGLE on blockchain by the time cost of state modification, because it can comprehensively reflect the consumed resources, e.g., CPU computation, and the load and read for memory and disk, during state modification [7]. Please note that we do not explicitly distinguish modifying and maintaining state, since they are interwoven in **OP1-4**.

6.1. How do computing resources affect NURGLE?

NURGLE crafts leaf nodes that contain a common prefix with a target leaf node, and inserts the crafted leaf nodes to deepen the target leaf node and proliferate intermediate nodes, causing extra consumed resources in modifying and maintaining *MPT* (**OP1-4**) (§5.1). Besides, according to Lemma 1, for a leaf node, if NURGLE can craft another leaf node that contains a common prefix with the target node at the length of x, and then NURGLE can deepen the target leaf node to the layer $x+2$. Hence, the larger x that NURGLE can find out, the deeper the target node can be deepened (i.e., $x+2$). Therefore, to assess capability of NURGLE under the different cost of computing resources, we evaluate the required computing resources of the adversary for colliding the different lengths of the common prefix.

To uniform the comparison of consumed computing resources, we fix hardwares used to conduct NURGLE (e.g., single RTX3080 GPU), and estimate the required computing resources by utilizing the cost time for NURGLE to collide a desired prefix at different lengths. In our evaluation, we launch NURGLE to collide an *indexing*, i.e., `0x1111..1111`, for demonstration. Besides, our evaluation relies on a practical assumption that the adversary adopts a brute force strategy to collide the target prefix (cf. details in Appendix A.1) [47]. Under the fixed computing resources (e.g., single RTX3080 GPU), NURGLE can conduct 1.90 billion hash calculations per second, and we record the time cost until NURGLE successfully crafts the target prefix. Table 2 shows our experimental results. In Table 2, the first column lists the length of the desired prefix in hash collision, the second column lists the final data crafted by NURGLE, the third column lists the corresponding *indexing* derived by the crafted data in the second column, and the fourth column lists the time cost for NURGLE to collide the desired prefix. As a result, under a reasonable consumption of computing resources, the adversary can collide out a desired prefix at the length of 13, which will deepen the target node to the layer 15 (Lemma 1 in §5.1). Please note that, the adversary can shorten the time cost of NURGLE by just deploying more GPUs or switching more powerful GPUs.

Answer to RQ1: *Computing resources affect the length of the collided prefix for a target indexing crafted by* NURGLE. *Under a reasonable consumption of computing resources,* NURGLE *can craft a desired prefix at the length of 13, which can deepen the target node to the layer 15 in* MPT.

6.2. How does NURGLE threaten blockchains?

Estimating attack impact. Time cost of blockchain for modifying and maintaining its *MPT* (i.e., **OP1-4** in Table 1) linearly increases with the number of nodes in *MPT* involved in modifying and maintaining state. Hence, we can estimate the overhead raised by NURGLE in state modification and maintenance according to the number of nodes in *MPT* proliferated by NURGLE. Specifically, we assume that NURGLE deepens several leaf nodes, and Eq. 2 derives the overhead (i.e., F_{nurgle}) in updating and maintaining deepened leaf

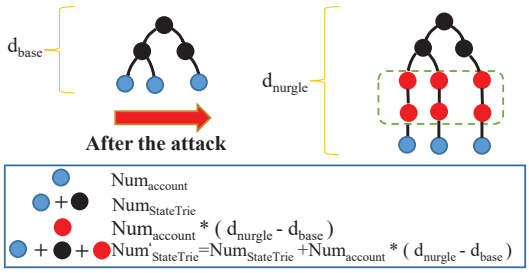


Figure 9: The nodes in *MPT* without and under the attack.

nodes in *MPT* brought by NURGLE. F_{nurgle} is obtained by dividing i) the number of nodes involved in handling (i.e., updating and maintaining) deepened leaf nodes in *MPT* under the attack by ii) the number of nodes involved in handling deepened leaf nodes in *MPT* without the attack. In Eq 2, $Num_{StateTrie}$ and $Num_{StorageTries}$ are denoted as the number of nodes involved in handling deepened leaf nodes in State Trie 3 and Storage Tries 4 without the attack, respectively. Besides, $Num'_{StateTrie}$ and $Num'_{StorageTries}$ are denoted as the number of corresponding nodes in State Trie 3 and Storage Tries 4 under the attack, respectively. Please note that, nodes in *MPT* can be partitioned into two parts, i.e., nodes in State Trie 3 and nodes in Storage Tries 4 (§2.2). Therefore, the number of nodes in *MPT* involved in handling deepened leaf nodes in *MPT* without and under the attack of NURGLE are $Num_{StateTrie}+Num_{StorageTries}$ and $Num'_{StateTrie}+Num'_{StorageTries}$, respectively.

$$F_{nurgle} = \frac{Num'_{StateTrie}+Num'_{StorageTries}}{Num_{StateTrie}+Num_{StorageTries}} \quad (2)$$

$$Num'_{StateTrie} = Num_{StateTrie}+Num_{Account} \times (d_{nurgle}-d_{base}) \quad (3)$$

$$Num'_{StorageTries}=Num_{StorageTries}+Num_{Slot} \times (d_{nurgle}-d_{base}) \quad (4)$$

The number of nodes involved in handling deepened leaf nodes in *MPT* can be trivially obtained by traversing *MPT*. In addition, we also present models to estimate $Num'_{StateTrie}$ (Eq. 3) and $Num'_{StorageTries}$ (Eq. 4), which can be used by adversaries to estimate the attack impact by Eq. 2 before launching NURGLE. Specifically, the number of nodes involved in handling deepened leaf nodes under the attack is the sum of i) the number of corresponding nodes without the attack, and ii) the number of proliferated intermediate nodes involved in handling deepened leaf nodes. Besides, the above proliferated intermediate nodes can be derived by the product of i) the number of deepened leaf nodes (i.e., $Num_{Account}$ in State Trie 3 and Num_{Slot} in Storage Tries 3), and ii) the number of layers that the leaf nodes are deepened (i.e., d_{nurgle}-d_{base}). In Eq. 3 and Eq. 4, d_{nurgle} and d_{base} refer to the layers where corresponding leaf nodes are located after and without the attack, respectively. For Fig. 9 as an example, three leaf nodes are involved in state modification in the State Trie 3, and NURGLE deepens them by two layers. Without the attack, $Num_{StateTrie}$ equals to 6. Besides, $Num_{Account}$ equals to 3, and d_{nurgle}-d_{base} equals to 2. Hence, according to Eq. 3, $Num'_{StateTrie}$ equals to $6 + 3 \times 2$, i.e., 12. It means that NURGLE causes 200% overhead (12/6) for maintaining and updating the three leaf nodes.

Evaluating attack impact. We evaluate attack impact in the fork of Ethereum, because it is the most popular blockchain under threats of NURGLE. Specifically, we synchronize an instrumented client [46] to capture the transactions of Ethereum from the block height of #14.99M to #15M (i.e., 10,000 blocks). By replaying captured transactions in corresponding blockchain state [48]–[50], we retrieve the nodes in *MPT* involved in state modification in the transactions. Specifically, we retrieve 7,116,002 nodes from State Trie 3 and 6,506,806 nodes from Storage Tries 4, which contain 712,565 and 2,041,719 leaf nodes, respectively. We then launch NURGLE in our forked *MPT* of Ethereum at the block height of #15M to deepen the retrieved leaf nodes to the layer 15. Finally, we traverse *MPT* to count how many intermediate nodes are proliferated by NURGLE, which will be involved in the state modification of previously retrieved leaf nodes. As a result, we find that the number of proliferated intermediate nodes is 1.11 times of all the retrieved nodes. It means that, under the attack, the time of state modification from #14.99M to #15M consumes more 111% resources, because it requires handling more 111% nodes proliferated by NURGLE. Besides, for all future transactions involving the nodes added by NURGLE, the time cost of their state modification is persistently raised. Please note that the majority of blockchain execution time is consumed by interactions with *MPT* (§4.1). Hence, the increase in the number of nodes involved in *MPT* can raise a considerable overhead on the execution performance of the blockchain, leading to performance degradation in the blockchain.

We further validate whether we can successfully estimate the attack impact. Specifically, the values of the six parameters (i.e., $Num_{Account}$, $Num_{StateTrie}$, Num_{Slot}, $Num_{StorageTries}$ d_{base}, and d_{nurgle}) from our captured transactions are 712,565, 7,116,002, 2,041,719, 6,506,806, 9.5, and 15, respectively. To simplify the estimation of attack impact, we have averaged out the d_{base}. Based on the six parameters, we derive that the corresponding F_{Nurgle} equals to 2.112. Our models estimate that there are more 111.2% nodes proliferated by NURGLE, causing 111.2% more consumed resources in handing *MPT*. Our results validate that our models estimate the impact of the attack almost perfectly (difference $< 1\%$).

Answer to RQ2: NURGLE *significantly degrades the execution performance of blockchain.*

6.3. How much does NURGLE cost?

The cost of leveraging NURGLE is an essential metric for a financially rational adversary. According to §3, the cost of NURGLE (denoted as G_{nurgle}) consists of two parts, i.e., G_{gpu}, the cost of computing resources (mainly GPUs [51]) for the calculation of hash collision (§5.1), and G_{gas}, the cost of gas fee for submitting attack payloads to blockchain via transactions. Besides, G_{gpu} is derived by the product of i) Num_{gpu}, the number of GPUs utilized by adversaries, ii) $Time_{hours}$, the hours of renting GPUs from GPU markets [52] by adversaries, and iii) $Price_{gpu}$, the USD price for renting

TABLE 3: The cost of NURGLE on seven different blockchains

Blockchain	$Price_{coin}$ (USD)	$Price_{gas}$	G_{gas} (USD)	G_{gpu} (USD)	G_{nurgle} (USD)	Optimized G_{gas} (USD)	Optimized G_{gpu} (USD)	Optimized G_{nurgle} (USD)
Ethereum	1,812	22.5 G wei	11,808,917.46	39.6	11,808,957.06	413,312.11	33	413,345.11
Binance Smart Chain	252.71	2.71 G wei	198,360.95	39.6	198,400.55	6,942.63	33	6,975.63
Heco	2.81	2.5 G wei	2,034.77	39.6	2,074.37	71.21	33	104.21
Polygon	0.71	206.30 G wei	42,596.22	39.6	42,635.82	1,490.86	33	1,523.86
Optimism	1,812	9.35×10^{-8} G wei	0.049	39.6	39.649	0.0017	33	33.0017
Avalanche	22.66	27.76 n AVAX	182,200.15	39.6	182,239.75	6,377.00	33	6,410.00
Ethereum Classic	16.52	1.17 G wei	5,596.96	39.6	5,636.56	195.89	33	228.89

Due to the volatility of cryptocurrency and gas prices, we have calculated their average values over a one-week period, spanning from June 6, 2023, to June 12, 2023.

a GPU in GPU markets. Please note that, when multiple GPUs are required in launching NURGLE, it is reasonable for adversaries to minimize the cost by renting GPUs from GPU markets (e.g., [52]) for a short period of time [51]. Furthermore, according to the specifications of gas mechanism [5], G_{gas} is derived by the product of i) $Price_{gas}$, the cryptocurrency (e.g., Ether) price of a unit of gas, ii) $Units_{gas}$, the units of consumed gas for executing the transactions containing attack payloads, and iii) $Price_{coin}$, the USD price of the cryptocurrency. We present corresponding equations for assessing the USD cost of NURGLE in Eq. 5 - Eq. 7.

In the rest of this section, we estimate the cost of NURGLE on seven mainstream blockchains (§6.3.1), and evaluate how the cost of NURGLE can be optimized (§6.3.2).

$$G_{nurgle} = G_{gas} + G_{gpu} \tag{5}$$

$$G_{gpu} = Num_{gpu} \times Time_{hours} \times Price_{gpu} \tag{6}$$

$$G_{gas} = Price_{gas} \times Units_{gas} \times Price_{coin} \tag{7}$$

6.3.1. Cost for attacking seven blockchains. We demonstrate NURGLE's attack towards seven popular blockchains (i.e., Ethereum, BSC, Heco, Polygon, Optimism, Avalance, and Ethereum Classic) [53], and measure corresponding attack cost of NURGLE. Since node numbers and layers in *MPT* of the seven blockchains are distinct, according to §6.1, the attack impact of NURGLE is different on the seven blockchains. To uniformly and fairly compare the cost of NURGLE on different blockchains, we fix the attack impact of NURGLE on different blockchains. Specifically, we reuse the attack impact for the attack launched by us in §6.2 on each blockchain to measure the cost of launching NURGLE.

We further derive actual values for parameters in Eq. 5 - Eq. 7 on seven blockchains in the following. Since the attack impact is fixed, the corresponding attack procedure of NURGLE should also be fixed, e.g., the procedure of hash collision (§5.1). Hence, for the same attack impact on different blockchains, the cost of computing resources (i.e., G_{gpu}) is the same. Specifically, according to §6.2, there are 2,754,284 (712,565 + 2,041,719) leaf nodes in *MPT* to be collided for being deepened by NURGLE. To conduct the hash collision for the leaf nodes, according to Lemma 2 and experimental results in §6.1, adversaries need to rent 33 RTX3080 GPUs for a period of 12 hours at least (cf. details in Appendix B.2). We obtain the corresponding price of GPU rental from [52], i.e., 0.1 USD/hour for renting a GPU. Therefore, to launch the attack in §6.2, G_{gpu} is 39.6 USD ($0.1 \times 33 \times 12$). Furthermore, since the seven blockchains are compatible with Ethereum and adopt the same gas mechanism, the units of gas consumed for executing the transactions containing attack payloads are also the same. Specifically, during the attack in §6.2, after forwarding attack payloads to the agent contract 5, the agent contract 5 executes its logic to insert crafted leaf nodes (§5.1) deepening the target 7,425,484 leaf nodes to the layer 15. As a result, it costs 289,647,227,381 units of gas. We further acquire the price of cryptocurrency (i.e., $Price_{coin}$) and the cryptocurrency price of a unit of gas ($Price_{gas}$) of each blockchain from corresponding dashboards (e.g., [53], [54]). Since $Price_{coin}$ and $Price_{gas}$ are volatile over time, we average out them in a period of one week.

Based on the derived parameters in Eq. 5 - Eq. 7, we list the detailed cost of NURGLE on different blockchain in Table 3. The second and third columns list the price of cryptocurrency and the cryptocurrency price of a unit of gas for each blockchain. For example, $Price_{coin}$ of Optimism is 1,812 USD, and $Price_{gas}$ of Optimism is 9.35×10^{-8}. The fourth, fifth, and sixth columns list G_{gas}, G_{gpu}, and G_{NURGLE} for launching NURGLE on different blockchain. For example, to degrade the performance of Optimism to 47% of original performance for a period of 10,000 blocks (§6.2), it only costs 39.64 USD for launching NURGLE.

6.3.2. Cost optimization. According to Table 3, the high gas fee leads to the expensive cost, e.g., the cost is over 11M USD on Ethereum. Inspired by active accounts (§5.1), we adopt an optimized strategy to decrease the cost of NURGLE by only deepening leaf nodes associated with active accounts, costing less and achieving a trade-off between attack impact and cost of NURGLE. Active accounts are accounts conducting frequent trades over a period of time (§5.1), hence, the leaf nodes associated with them are the most frequently modified and accessed leaf nodes in *MPT*.

We further inspect the 2,754,284 leaf nodes deepened by NURGLE during the attack in §6.2. It shows that 2,103,558 of them (i.e., 361,703 and 1,741,855 leaf nodes in State Trie 3 and Storage Tries 4) are only modified and accessed once among transactions of the 10,000 blocks (§6.2). Compared with the leaf nodes, the other 650,726 leaf nodes (2,754,284-2,103,558) have been collectively modified and accessed 5,321,926 times. Hence, if NURGLE only deepens the 23.63% (650,726/2,754,284) leaf nodes, the attack impact of NURGLE retains 71.67% (5,321,926/(2,103,558+5,321,926)) of original attack impact. Besides, NURGLE's G_{gas} is only 19.61% of original G_{gas} (§4.1). Please note that, the cost of deepening leaf nodes in State Trie 3 and Storage Tries 4 is different

TABLE 4: Cost optimization based on active accounts

Count	Retained impact	Retained cost	Optimized G_{gas} (USD)
1	100.00%	100.00%	11,808,917.46
2	71.67%	19.60%	2,314,547.82
4	58.16%	5.76%	680,193.64
6	54.66%	3.50%	413,312.11

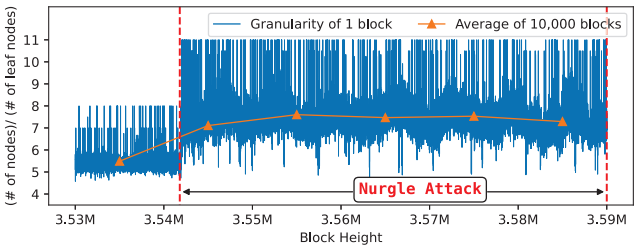
(Appendix B.1). We enumerate other cases in Table 4 for launching NURGLE on Ethereum. For example, if NURGLE only deepens the leaf nodes modified and accessed no less than 6 times, G_{gas} of NURGLE is 3.5% of original G_{gas}, and the retained impact is 54.66% of original attack impact. As a result, we provide the optimized cost (e.g., G_{gas}) of NURGLE on the seven blockchains in Table 3 in the cases that NURGLE only deepens the leaf nodes associated with the active accounts modified and assessed no less than 6 times. Since the number of leaf nodes deepened by NURGLE decreases, the corresponding G_{gpu} also reduces to 33 USD accordingly (Appendix B.2). It shows that, by optimized strategies based on active accounts, the cost of NURGLE on Polygon reduces from 42,596.22 USD to 1,523.86 USD, retaining 54.66% of original attack impact.

Baseline comparison. We compare the cost of creating spam transactions that result in the same attack impact as NURGLE with optimized strategies. Specifically, for each block within the range of #14.99M to #15M, we iteratively submitted transactions sampled from Ethereum to the block, ensuring that the number of *MPT* nodes handled in the submitted transactions aligned with the number of *MPT* nodes proliferated by NURGLE. As a result, the baseline costs 3,826,037.45 USD on Ethereum, which is 9.25 times higher than NURGLE with optimized strategies in Table 3.

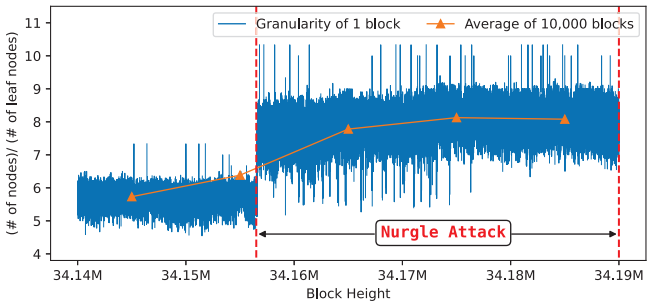
Answer to RQ3: NURGLE*'s cost depends on the gas fee, and is reasonable for most cases, and our optimization strategy further minimizes the cost of* NURGLE.

6.4. Can NURGLE threaten blockchain in practice?

Compared to demonstrating NURGLE in controlled environments [6], we choose to determine whether NURGLE can threaten the current blockchain in practice. Please note that, unlike previous studies [2], [25], [26] that only have non-persistent attack impact, the impact of NURGLE will be persistent in the blockchain. This is because NURGLE persistently proliferates the intermediate nodes in *MPT*, and persistently exacerbates resource consumption for the maintaining and updating of *MPT*. Therefore, due to ethical concerns and inspired by previous studies [2], [26], we choose to launch NURGLE on blockchain testnets, i.e., Ethereum Sepolia testnet [55] and BSC testnet [56]. Please note that testnets set the closest environment to the practice, and it is built for researchers and developers to conduct experiments without risk to real funds or the main chain [57]. Considering that there are other developers and researchers who are active in the testnet, we further minimize the potential ethical issues by carefully adjusting attack parameters from scratch to light the attack impact on testnets.



(a) The tendency of involved MPT nodes of Ethereum testnet.



(b) The tendency of involved MPT nodes of BSC testnet.

Figure 10: During the exploitation, NURGLE proliferates intermediate nodes in *MPT*, and deepens leaf nodes, leading to an increase in the number of *MPT* nodes to be modified to update a leaf node. NURGLE can cause about 32% and 39.4% more *MPT* nodes to be involved in state modification per block in Ethereum testnet and BSC testnet, respectively.

In our evaluation, we synchronize blockchain clients to obtain latest state of the two testnets. We launch NURGLE to exploit the two testnets by following the seven steps of NURGLE (§5.2). Specifically, NURGLE first extracts the list of all accounts 1 in the two testnets and *MPT* of two testnets 2. We next leverage indexing retriever 3 to retrieve the target leaf nodes and their *indexing*. After computing unit 4 crafts the leaf nodes to be inserted for proliferating the intermediate nodes, we forward the crafted data to our agent contracts (i.e., 0xc8f2...199d in Ethereum testnet and 0xc062...a163 in BSC testnet) 5 to finalize the attack.

We launched the attack of NURGLE on the Ethereum (resp. BSC) testnet at the block height of #3,541,798 (May 23, 2023) (resp. #34,156,452 (Oct. 13, 2023)), and ceased the attack at the block height of #3.59M (Jun. 1, 2023) (resp. #34.17M (Oct. 13, 2023)). During the whole exploitation of NURGLE, we forwarded attack payloads to agent contracts 5 by 330 (resp. 53) transactions, and we inserted 100 leaf nodes into State Trie 3 in *MPT* for each transaction. Fig. 10 shows our experimental results, and it depicts the tendency of the number of nodes in *MPT* to be modified to update a leaf node. Specifically, during the exploitation, NURGLE proliferates intermediate nodes in *MPT*, and deepens leaf nodes, leading to an increase in the number of nodes in *MPT* to be modified to update a leaf node. As mentioned in §4.1, the consumed resources (e.g., the time cost) for state modification (e.g., **OP1-4**) linearly increases with the number of involved nodes. Hence, Fig. 10 indicates that the state modification is significantly exacerbated by the

exploitation of NURGLE. To obtain a comprehensive understanding of the impact of NURGLE's exploitation on the testnet, we further investigate the performance overhead of the blockchain brought by NURGLE. As a result, during the attack against Ethereum (resp. BSC) testnet, 32% (resp. 39.4%) more nodes in *MPT* are involved in state modification per block, besides, NURGLE raises the cost time of state modification by 15% (resp. 18%). We further evaluate the overall performance degradation caused by NURGLE by using the Metrics module [58], which enables us to collect execution information about blockchain clients. As a result, NURGLE results in a 10.7% (resp. 12.4%) rise in end-to-end execution time of Ethereum (resp. BSC) testnet.

Answer to RQ4: NURGLE *threatens blockchain by causing more nodes involved in state modification, and raises the cost time of the overall blockchain execution.*

7. Discussion

7.1. Practical attack impact of NURGLE

NURGLE can threaten blockchain in seven aspects.

i) As the ever-evolving blockchains [20], attack scenarios of NURGLE are extremely rich, especially the emerging blockchains. Taking a newly deployed blockchain as an example, and assuming that its leaf nodes in *MPT* are in the layer 5, NURGLE can deepen a leaf node to the layer 15 (§6.1), thereby proliferating extra 10 intermediate leaves for the blockchain to updating the leaf node.

ii) The impact of NURGLE on blockchain is persistent, and the victim blockchain will be impacted by manipulated *MPT* in all subsequent blocks by the nodes proliferated by NURGLE. Besides, according to §6.1, the attack impact of NURGLE can be further exacerbated by adversaries with more powerful computing resources (e.g., GPUs).

iii) NURGLE delays users in using blockchain and AUX (e.g., flashbot [59], [60], infura [61], ENS [62]) in providing services, because NURGLE exacerbates resource consumption of blockchain, and increases the time cost for maintaining and updating *MPT*. For example, an AUX like infura can only provide their services after they finish the delay of updating the latest state in *MPT*.

iv) Since NURGLE can delay the processing of user transactions, attackers can launch the delay attacks [63] to threaten the liveness of the layer 2 rollups. Specifically, attackers can utilize NURGLE to slow down the confirmation of the transactions for verifying the validity of layer 2 transactions.

v) NURGLE threatens the consensus security of blockchain by increasing the execution costs of running blockchain nodes, which subsequently results in reducing the number of nodes participating in the blockchain network [26].

vi) NURGLE erodes trust in the affected blockchains, leading to a decline in the value of their cryptocurrencies [25].

vii) The overhead of blockchain execution raised by NURGLE (e.g., the 10.7% overall performance degradation in §6.4) can waste the energy of all blockchain nodes.

As mentioned in §6.1, computing resources (i.e., GPUs) of adversaries influence the attack impact of NURGLE. Concretely, computing resources affect the depth of target leaf nodes in *MPT* deepened by NURGLE, and the number of intermediate nodes proliferated by NURGLE, which finally impacts how much the resource consumption in *MPT* will be exacerbated to impair the blockchain's performance. Our evaluation on NURGLE suffers some limitations that we do not explore the best attack impact of NURGLE, because we choose to evaluate the attack impact of NURGLE under a reasonable resource cost of NURGLE (§6.1). Hence, our experimental results should be considered as the lower bound of the potential attack impact brought by NURGLE.

7.2. On-demand attacks of NURGLE

Instead of one-time short exploitation, it is also feasible to control NURGLE on demand. During the on-demand attacks, attackers can craft several deeper leaf nodes controlled by themselves in advance, and only update it to slow down transactions when needed. We further evaluate the impact and cost of on-demand attacks by examining two distinct on-demand attacks, each targeting users of different contracts. Our experimental results demonstrate that on-demand attackers can slow down user transactions of specified contracts by crafting attack payloads in advance. Besides, while implementing attacks with the same impact (e.g., slowing down all user transactions), the attack cost depends on the logic of involved contracts. In the following, we elaborate on how we evaluate the two on-demand attacks and the detailed experimental results.

- In the first attack, the adversary slows down all users of an AMM (Automated Market Maker) contract [64], where users can exchange two specific tokens with the AMM contract. Since the token balances of the AMM contract will update when users interact with it, the adversary can deepen the two leaf nodes storing the token balances of the AMM contract for the two tokens to slow down all user transactions. The cost of transaction fees is 4.06 USD on Ethereum, and the cost of computing resources is 4.32 USD.
- In the second attack, the adversary delays all users of a token contract. Since, for each transaction, only the token balances of the users in the current transaction will update, the adversary needs to deepen all the leaf nodes storing all users' token balances to slow down all user transactions. In our experiments, the token contract has 10,000 users. Hence, the cost of transaction fees is 20,154.95 USD on Ethereum, and the cost of computing resources is 23.56 USD.

7.3. Feasible mitigations against NURGLE

In the following, we detail the three feasible mitigations, and discuss their advantages and disadvantages.

i) **Verkle tree** [65]. Verkle tree mitigates the impact of NURGLE by indexing fewer nodes in its tree structure and adopting a swifter authenticated method [65]. First, compared with the prefix tree in MPT structure, the structure of verkle tree is designed to be flatter, which compresses the distance

from the leaf node to the root node. The verkle tree alleviates the volume of nodes to be updated and verified associated with **OP1** and **OP2** (§4.1). Second, verkle tree adopts a polynomial commitment scheme [66] rather than the hash-style vector commitment of MPT structure [5]. During the verification for the latest state, the polynomial commitment reduces the number of nodes to be verified [66]. Verkle tree has the advantage of consuming fewer resources when accessing a leaf node by involving fewer intermediate nodes. However, the advantage comes at the cost of increased space wastage in Verkle tree, as each node consumes more space. For instance, a branch node in Verkle tree, which includes 256 pointers, occupies nearly 16 times more space than a branch node in *MPT*, which only has 16 pointers.

ii) **Trimming historical state (EIP-4444)** [67]. EIP-4444 eases NURGLE's impact by periodically reducing *MPT*'s size. Specifically, *MPT* will undergo pruning by retaining the state of blockchain for nearly one year. Ethanos [8] has carried out its implementation in a similar manner. Trimming historical state has the advantage of reducing the storage space required by a blockchain node by eliminating a portion of its historical state. However, the mitigation comes with the disadvantage that the blockchain node cannot conduct complete security verification of the whole blockchain state due to the elimination of the historical state.

iii) **New patch for gas mechanism**. We plan to propose a new Ethereum improvement proposal (EIP) for the gas mechanism to defend against NURGLE. In our EIP, gas fee of transactions will be equivalent to actual consumed resources, e.g., considering the consumed resources for modifying nodes in *MPT*. The mitigation has the advantage of implementing a fairer gas mechanism. The adjusted mechanism ensures that the gas fee for transactions corresponds directly to the actual resources consumed when modifying nodes in the MPT. However, the mitigation can introduce new attack vectors. For instance, an adversary can exploit the adjusted mechanism by proliferating intermediate nodes to popular contracts, thereby increasing gas fees for all users of the contracts. Hence, it is crucial to examine the implementation of the third mitigation thoroughly.

7.4. Vulnerability disclosure

At the time of writing, 588 blockchains compatible with the Ethereum ecosystem (§2) are directly under the threats of NURGLE. Besides, we reveal that other 153 blockchains compatible with the Polkadot ecosystem [22] are also threatened by NURGLE, because Polkadot also adopts the MPT structure to handle state storage similar to Ethereum (§2). However, there are two major differences between Polkadot and Ethereum. i) Hash algorithm. Polkadot utilizes xxhash algorithm [68] instead of keccak256 algorithm [23] in their blockchain design, e.g., they use xxhash to derive the *indexing* of its leaf nodes. ii) Fee mechanism. Polkadot adopts the weight-based mechanism to cost fee for transactions rather than gas mechanism adopted in Ethereum. Concretely, transaction fee is adjusted according to the congestion cost of the block. We enumerate the above vulnerable blockchains under NURGLE in Appendix C.1. We will explore the hash collision of NURGLE (§5.1) against xxhash algorithm and optimize attack parameters against the fee mechanism of Polkadot to further investigate the vulnerability of Polkadot under NURGLE as our future work.

Ethics concerns. We reported vulnerabilities brought by NURGLE to seven mainstream blockchains with high market capitalization before the paper submission. At the time of writing, we have received responses from six of them, confirming the vulnerabilities and rewarding us with thousands of USD bounty. Furthermore, before the publication, we reported the vulnerabilities to all other affected blockchains. Currently, an additional 16 teams from the newly reported blockchain teams have responded with positive acknowledgments. We present the details of their responses in Appendix C. Besides, we will release corresponding materials 90 days before the publication following ethical obligations and conference committee requirements.

8. Related work

In this section, we explore closely related studies in four aspects by the attack surfaces of DoS attacks on blockchain.

Consensus network. Based on the fault-tolerant mechanism, consensus network assists blockchain nodes in achieving an agreement on the latest state. Compromising consensus network incurs the blockchain to violate its consensus functionalities [69]. Heo et al. [26] achieve the isolation and disconnection of an Ethereum node from main network by hijacking half of peer connections on the blockchain network. Tran et al. [70] propose an attack that isolates Bitcoin nodes via malicious Internet Service Providers. Chen et al. [69] detect vulnerabilities that cause denial of service in the consensus network through fuzz testing [71]. Prunster et al. [72] mount eclipse attack [73] on Inter Planetary File System (IPFS) by poisoning the node's routing information, so that the node is isolated from the main network. Yang et al. [4] unveil a vulnerability that can collapse the consensus of the blockchain. The root cause of the vulnerability is that the state of the blockchain client implementation in different languages is inconsistent. Saad et al. [74] propose SyncAttack which uses fluctuations in the Bitcoin network to achieve blockchain forks. SDoS [75] is a DoS attack based on selfish mining. They find that an adversary can launch a 51% attack [76] to destroy the consensus of the blockchain with only 19.6% of the computing power.

Txpools. Txpools maintain pending transactions from blockchain users, and miners/validators will pack transactions from their txpools to blockchain. Adversaries sway the security of blockchain by interfering with the transaction packing involved in txpools. Deter [2] paralyzes Ethereum transaction pool by crafting malicious transactions, and prevents users from interacting with the blockchain. Wu et al. [77] propose a distributed denial-of-service (DDoS) attack against the Bitcoin mining pool with the idea of game theory. Poster [78] denies the memory pool of Bitcoin and traps users into paying higher mining fees. Yaish et al. [79] leverage the censorship mechanism adopted in Ethereum to

craft three classes of DoS attacks, resulting in burdening victims' computational resources and clogging their txpools.

Auxiliary services. Auxiliary services refer to entities facilitating blockchain's efficiency at off-chain. DoS attackers can refuse users to utilize services provided by AUX. Li et al. [61] propose a DoS attack against RPC services in Ethereum. It indirectly causes users to be unable to access the Ethereum mainnet by paralyzing the RPC service. Nguyen et al. [80] discover a flood attack against a single shard, which undermines the performance of blockchain. They present a scheme by the Trusted Execution Environment (TEE) to counter the flood attack.

Smart contracts. Smart contracts are automation programs executed in blockchain. One of their security issues is underpriced opcodes [81], whose gas cost is substantially less than the consumed resources. Attackers can lead blockchain to consume extremely high resources while executing underpriced opcodes. Chen et al. [81] reveal the under-priced opcodes on Ethereum, and attackers can wield these opcodes to launch DoS attacks. They propose an adaptive gas mechanism to defend against the DoS attacks mentioned by them by auto-adjusting the gas fee of opcodes. Perez et al. [25] use genetic methods [82] to generate smart contracts with low gas and high resource consumption, by leveraging the under-priced instructions of Ethereum. eTainter [83], [84] inspects the DoS attack based on the gas mechanism of smart contracts through static program analysis. In addition, the design of NURGLE is also motivated by existing studies that exploit historical state [25], [61], [81].

9. Conclusion

We reveal a new attack surface, i.e., state storage, of blockchains. Besides, we present NURGLE, the first DoS attack exploiting state storage. By proliferating intermediate nodes within state storage, NURGLE forces blockchains to expend additional resources on state maintenance and verification, impairing their performance. We further conduct a comprehensive and systematic evaluation of NURGLE. Experimental results show that NURGLE can significantly degrade execution performance of blockchains, under a reasonable financial cost. Understanding and mitigating the threats brought by NURGLE are crucial for ensuring the stability and resilience of blockchain ecosystems. Our contributions shed light on the importance of securing the state storage in blockchain, encouraging further research and development to safeguard against NURGLE and similar attacks.

Acknowledgements The authors thank anonymous reviewers for their constructive comments. This work is partly supported by Hong Kong RGC Projects (No. PolyU15222320), National Natural Science Foundation of China under Grant No. 62332004, U22B2029 and U19A2066, the Natural Science Foundation of Sichuan Province under Grant 2022NSFSC0871, and the Financial Support for Outstanding Talents Training Fund in Shenzhen.

References

[1] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, and A. Gervais, "Sok: Decentralized finance (defi) incidents," in *SP*, 2023.

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

[3] "Geth security release: Critical patch for cve-2020-28362, november 2020." https://blog.ethereum.org/2020/11/12/geth_security_release.

[4] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *OSDI*, 2021.

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

[6] C. Li, S. M. Beillahi, G. Yang, M. Wu, W. Xu, and F. Long, "LVMT: An efficient authenticated storage for blockchain," in *OSDI*, 2023.

[7] S. Ponnapalli, A. Shah, S. Banerjee, D. Malkhi, A. Tai, V. Chidambaram, and M. Wei, "Rainblock: Faster transaction processing in public blockchains." in *USENIX ATC*, 2021.

[8] J.-Y. Kim, J. Lee, Y. Koo, S. Park, and S.-M. Moon, "Ethanos: efficient bootstrapping for full nodes on account-based blockchain," in *EuroSys*, 2021.

[9] Y. Kim, S. Jeong, K. Jezek, B. Burgstaller, and B. Scholz, "An off-the-chain execution environment for scalable testing and profiling of smart contracts." in *USENIX ATC*, 2021.

[10] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *SP*, 2022.

[11] C. Yue, Z. Xie, M. Zhang, G. Chen, B. C. Ooi, S. Wang, and X. Xiao, "Analysis of indexing structures for immutable data," in *SIGMOD*, 2020.

[12] "gas mechanism," https://ethereum.org/en/developers/docs/gas/.

[13] P. Ruan, T. T. A. Dinh, D. Loghin, M. Zhang, G. Chen, Q. Lin, and B. C. Ooi, "Blockchains vs. distributed databases: Dichotomy and fusion," in *SIGMOD*, 2021.

[14] Q. Wei, B. Li, W. Chang, Z. Jia, Z. Shen, and Z. Shao, "A survey of blockchain data management systems," *TECS*, 2022.

[15] "Bnb smart chain," https://www.bnbchain.org/.

[16] "Polygon," https://polygon.technology/.

[17] "Avalanche," https://www.avax.network/.

[18] "Optimism," https://www.optimism.io/.

[19] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," *White paper*, 2016.

[20] "The ethereum ecosystem," https://chainlist.org/.

[21] X. Yi, Y. Fang, D. Wu, and L. Jiang, "Blockscope: Detecting and investigating propagated vulnerabilities in forked blockchain projects," in *NDSS*, 2023.

[22] "The polkadot ecosystem," https://substrate.io/ecosystem/projects/.

[23] I. Dinur, O. Dunkelman, and A. Shamir, "New attacks on keccak-224 and keccak-256," in *Fast Software Encryption*, 2012.

[24] P. Zheng, Z. Zheng, X. Luo, X. Chen, and X. Liu, "A detailed and real-time performance monitoring framework for blockchain systems," in *ICSE-SEIP*, 2018.

[25] D. Perez and B. Livshits, "Broken metre: Attacking resource metering in evm," *NDSS*, 2019.

[26] H. Heo, S. Woo, T. Yoon, M. S. Kang, and S. Shin, "Partitioning ethereum without eclipsing it," in *NDSS*, 2023.

[27] M. Ayub, T. Saleem, M. Janjua, and T. Ahmad, "Storage state analysis and extraction of ethereum blockchain smart contracts," *TOSEM*, 2023.

[28] "Popular contract from dappradder," https://dappradar.com/rankings.

[29] W. Szpankowski, "Patricia tries again revisited," *JACM*, 1990.

[30] M. Szydlo, "Merkle tree traversal in log space and time," in *Eurocrypt*, 2004.

[31] H. XiaoJu, G. XueQing, H. ZhiGang, Z. LiMei, and G. Kun, "Ebtree: A b-plus tree based index for ethereum blockchain data," in *ASSE*, 2020.

[32] P. A. Saffi and K. Sigurdsson, "Price efficiency and short selling," *The Review of Financial Studies*, 2011.

[33] "State prune technology," https://blog.ethereum.org/2015/06/26/state-tree-pruning.

[34] P. Ramya and T. SairamVamsi, "Securing manets using sha3 keccak algorithm," in *ICICCT*, 2020.

[35] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, 2010.

[36] "create2crunch," https://github.com/0age/create2crunch.

[37] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? application to des," in *EUROCRYPT*, 1989.

[38] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," in *CRYPTO*, 2003.

[39] W. Xu and A. K. Tang, "A generalized coupon collector problem," *Journal of Applied Probability*, 2011.

[40] "solidity doc," https://docs.soliditylang.org/en/v0.8.21/.

[41] T. Chen, Z. Li, X. Luo, X. Wang, T. Wang, Z. He, K. Fang, Y. Zhang, H. Zhu, H. Li, and X. Zhang, "Sigrec: Automatic recovery of function signatures in smart contracts," *IEEE TSE*, 2021.

[42] K. Zhao, Z. Li, J. Li, H. Ye, X. Luo, and T. Chen, "Deepinfer: Deep type inference from smart contract bytecode," in *ESEC/FSE*, 2023.

[43] V. Fabian and B. Vitalik, "Eip-20: Token standard," https://eips.ethereum.org/EIPS/eip-20, 2015.

[44] Z. He, S. Song, Y. Bai, X. Luo, T. Chen, W. Zhang, P. He, H. Li, X. Lin, and X. Zhang, "Tokenaware: Accurate and efficient bookkeeping recognition for token smart contracts," *TOSEM*, 2023.

[45] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019, pp. 1503–1520. [Online]. Available: https://doi.org/10.1145/3319535.3345664

[46] "go-ethereum," https://github.com/ethereum/go-ethereum/releases/tag/v1.11.6.

[47] J. Kelsey and T. Kohno, "Herding hash functions and the nostradamus attack," in *EUROCRYPT*, 2006.

[48] L. Zhou, K. Qin, and A. Gervais, "A2mm: Mitigating frontrunning, transaction reordering and consensus instability in decentralized exchanges," *arXiv*, 2021.

[49] S. Li, J. Li, Y. Tang, X. Luo, Z. He, Z. Li, X. Cheng, Y. Bai, T. Chen, Y. Tang, Z. Liu, and X. Zhang, "Blockexplorer: Exploring blockchain big data via parallel processing," *TC*, 2023.

[50] T. Chen, Z. Li, Y. Zhang, X. Luo, A. Chen, K. Yang, B. Hu, T. Zhu, S. Deng, T. Hu, J. Chen, and X. Zhang, "Dataether: Data exploration framework for ethereum," in *ICDCS*, 2019.

[51] G. Liu, D. Liu, S. Hao, X. Gao, K. Sun, and H. Wang, "Ready raider one: Exploring the misuse of cloud gaming services," in *CCS*, 2022.

[52] "Gpu market," https://vast.ai/.

[53] "coinmarketcap," https://coinmarketcap.com/.

[54] "gas fees," https://dune.com/soispoke/l1-cross-chain-gas-fees.

[55] "ethereum sepolia testnet," https://sepolia.etherscan.io/.

[56] "bsc testnet," https://testnet.bscscan.com/.

[57] "testnet-vs-mainnet," https://shardeum.org/blog/testnet-vs-mainnet/.

[58] "Geth metrics module," https://geth.ethereum.org/docs/monitoring/metrics.

[59] B. Weintraub, C. F. Torres, C. Nita-Rotaru, and R. State, "A flash (bot) in the pan: measuring maximal extractable value in private pools," in *IMC*, 2022.

[60] Z. Li, J. Li, Z. He, X. Luo, T. Wang, X. Ni, W. Yang, C. Xi, and T. Chen, "Demystifying defi mev activities in flashbots bundle," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[61] K. Li, J. Chen, X. Liu, Y. R. Tang, X. Wang, and X. Luo, "As strong as its weakest link: How to break blockchain dapps at rpc service." in *NDSS*, 2021.

[62] P. Xia, H. Wang, Z. Yu, X. Liu, X. Luo, G. Xu, and G. Tyson, "Challenges in decentralized name management: The case of ens," in *IMC*, 2022.

[63] "Delay attacks on rollups," https://ethresear.ch/t/delay-attacks-on-rollups/14508.

[64] J. Xu, N. Vavryk, K. Paruch, and S. Cousaert, "Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols," *ACM Computing Surveys*, 2023.

[65] J. Kuszmaul, "Verkle trees," *Verkle Trees*, 2019.

[66] D. Boneh, J. Drake, B. Fisch, and A. Gabizon, "Halo infinite: Proof-carrying data from additive polynomial commitments," in *CRYPTO*, 2021.

[67] "Eip-4444," https://eips.ethereum.org/EIPS/eip-4444.

[68] "xxhash: Extremely fast hash algorithm," https://github.com/Cyan4973/xxHash.

[69] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *SP*, 2022.

[70] 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 *SP*, 2020.

[71] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *SP*, 2018.

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

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

[74] M. Saad, S. Chen, and D. Mohaisen, "Syncattack: double-spending in bitcoin without mining power," in *CCS*, 2021.

[75] Q. Wang, T. Xia, D. Wang, Y. Ren, G. Miao, and K.-K. R. Choo, "Sdos: Selfish mining-based denial-of-service attack," *IEEE TIFS*, 2022.

[76] C. Badertscher, Y. Lu, and V. Zikas, "A rational protocol treatment of 51% attacks," in *CRYPTO*, 2021.

[77] S. Wu, Y. Chen, M. Li, X. Luo, Z. Liu, and L. Liu, "Survive and thrive: A stochastic game for ddos attacks in bitcoin mining pools," *IEEE/ACM Transactions on Networking*, 2020.

[78] M. Saad, M. T. Thai, and A. Mohaisen, "Poster: deterring ddos attacks on blockchain-based cryptocurrencies through mempool optimization," in *CCS*, 2018.

[79] A. Yaish, K. Qin, L. Zhou, A. Zohar, and A. Gervais, "Speculative denial-of-service attacks in ethereum," *ePrint*, 2023.

[80] T. Nguyen and M. T. Thai, "Denial-of-service vulnerability of hash-based transaction sharding: Attack and countermeasure," *IEEE TC*, 2023.

[81] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks," in *ISPEC*, 2017.

[82] D. Whitley, "A genetic algorithm tutorial," *Statistics and computing*, 1994.

[83] A. Ghaleb, J. Rubin, and K. Pattabiraman, "etainter: detecting gas-related vulnerabilities in smart contracts," in *ISSTA*, 2022.

[84] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Analyzing the out-of-gas world of smart contracts," *CACM*, 2020.

[85] A. Joux and T. Peyrin, "Hash functions and the (amplified) boomerang attack," in *CRYPTO*, 2007.

Appendix A. Hash collision of NURGLE

A.1. The method of hash collision

In the following, we introduce how we collide the prefix of a target keccak256 hash value (i.e., *indexing* of a leaf node). We assume that adversaries do not have any advanced knowledge of cryptography, and they apply the most primitive brute force hash collision strategy [85]. Brute force hash collision refers to exhausting the results of all possible keccak256 hash calculations, until a keccak256 hash value is crafted and satisfies adversaries' requirements. Concretely, the adversaries first pick a keccak256 hash value, and then choose different inputs x_i, where $i \in \mathbb{N}^*$. After that, the adversaries check whether the prefix of the hash result of x_i matches the prefix of the picked keccak256 hash value.

Multi-target hash collision. Similarly, it does not rely on any advanced knowledge of cryptography. To conduct a multi-target hash collision, adversaries first pick ϕ target keccak256 hash values, and then choose different inputs x_i, where $i \in \mathbb{N}^*$. After that, the adversaries check whether the prefix of the hash result of x_i matches the prefix of one keccak256 hash value of the ϕ picked keccak256 hash values. The adversaries finalize the multi-target hash collision until all the ϕ picked keccak256 hash values are matched.

Appendix B. Estimating the cost of NURGLE

In this section, we will elaborate on how to estimate the cost of NURGLE before launching the attack. As mentioned in §6.2, the attack impact can also be estimated. Hence, the adversaries of NURGLE can strategically determine the trade-off of their actual attack, by firstly estimating the cost and attack impact of launching NURGLE.

B.1. The cost of units of gas

NURGLE needs to cost 289,647,227,381 units of gas during the attack (§6.3). Here we elaborate on how to estimate the cost of units of gas by Eq. 8 - 10. Besides, we reuse the same symbols in §6.3 and §6.2. Eq. 8 indicates that the units of gas consist of two parts, i.e., the units of gas consumed for inserting leaf nodes in State Trie 3 ($Gas_{StateTrie}$) and the units of gas consumed for inserting leaf nodes in Storage Tries 4 ($Gas_{StorageTries}$). According to §5.1, the units of gas consumed for inserting single leaf node in State Trie 3 and Storage Tries 4 are different. Eq. 9 and Eq. 10 adopt the same method to calculate the units of gas for $Gas_{StateTrie}$ and $Gas_{StorageTries}$. For Eq. 9 as an example, $Gas_{StateTrie}$ is obtained as the product of the total number of leaf nodes that NURGLE needs to insert (i.e., $Num^{insert}_{StateTrie}$) and the units of gas required to insert single leaf node in State Trie 3 (i.e., 21,000). The units of gas for inserting a single leaf node in State Trie 3 is 21,000 units of gas, because we insert a leaf node in State Trie 3 by sending 1 wei to a target account (§5.1) [5].

$$Units_{gas} = Gas_{StateTrie} + Gas_{StorageTries} \tag{8}$$

$$Gas_{StateTrie} = Num^{insert}_{StateTrie} \times 21000 \tag{9}$$

$$Gas_{StorageTries} = Num^{insert}_{StorageTries} \times Cost_{StorageTries} \tag{10}$$

The only difference between Eq. 9, and Eq. 10 is the cost of inserting a leaf node. This is because, according to §5.1, we insert a leaf node in Storage Tries 4 by invoking a function of a target contract, hence, we denote the units of gas cost by inserting a leaf node in Storage Tries 4 as $Cost_{StorageTries}$, e.g., if we invoke `transfer()` (§5.1), it costs 44258 units of gas for inserting a leaf node in *MPT*.

B.2. Cost of computing resources

According to §6.3, to deepen all 2,754,284 leaf nodes, we need 33 RTX3080 GPUs for the rental of 12 hours, i.e., 39.6 USD. Here we detail how to estimate the amount of GPUs and the required time. Besides, we reuse the same symbols in §5.1 and §6.1.

First, we need to confirm the number of leaf nodes (i.e., ϕ) whose *indexing* are required to be collided by NURGLE. We can then estimate the minimal number of hash collisions that are required to be satisfied for NURGLE. According to §6.2, and **S-3** in §5.1, the minimal number of hash collisions equals to $\phi \times \frac{d_{nurgle}\text{-}d_{base}}{2}$. We further adopt the multi-target hash collision strategy (§5.1) on multiple GPUs to collide the *indexing* of all $\phi \times \frac{d_{nurgle}\text{-}d_{base}}{2}$ targets. As mentioned in §5.1, we parallelized conduct the keccak256 hash calculations, and we denoted GPU_{time} as the time (hours) for

TABLE 5: Response status of seven mainstream blockchains

Blockchain	Response status
Ethereum	Accepted the vulnerabilities.
Binance Smart Chain	Accepted with the vulnerabilities, and rewarded us with bug bounty.
Heco	No reponse
Polygon	Accepted the vulnerabilities.
Optimism	Accepted the vulnerabilities.
Avalanche	Accepted with the vulnerabilities, and rewarded us with bug bounty.
Ethereum Classic	Accepted the vulnerabilities.

single GPU to finish the multi-target hash collision. We denote θ as the calculation counts for colliding one *indexing*, and P as the calculation counts that single GPU can finish in one hour. Hence, we can estimate GPU_{time} by Eq. 11. For example, according to §6.1, $\frac{\theta}{P}$ equals to 24.58 hours to deepen a leaf node to the layer 15. According to §6.3, there are 2,754,284 leaf nodes to be collided, i.e., ϕ equals to 2,754,284. According to §6.2, d_{nurgle} and d_{base} equal to 15 and 9.5, respectively. Hence, $ln(\phi \times \frac{d_{nurgle}-d_{base}}{2})$ equals to 15.84. As a result, GPU_{time} equals to 389.35 hours, i.e., single GPU needs 389.35 hours to collide the 2,754,284 leaf nodes. In other words, it costs about 33 GPUs to collide the 2,754,284 leaf nodes in 12 hours (389.35/33).

$$GPU_{time}=\frac{\theta}{P} \times ln(\phi \times \frac{d_{nurgle}-d_{base}}{2}) \tag{11}$$

Appendix C. Response status of blockchains

We have reported the vulnerabilities exploited by NURGLE to seven mainstream blockchain platforms, including Ethereum, Binance Smart Chain, Heco, Polygon, Optimism, Avalanche, and Ethereum Classic. In Table 5, we summarize their latest responses for the corresponding vulnerabilities. Specifically, six blockchains of them, i.e., Ethereum, Binance Smart Chain, Polygon, Optimism, Avalanche, and Ethereum Classic have accepted the vulnerabilities and were exploring appropriate countermeasures against NURGLE. Especially, we have received thousands of USD bounty from Binance Smart Chain and Avalanche. Furthermore, before the publication, we reported the vulnerabilities to all other affected blockchains. Currently, an additional 16 teams from the newly reported blockchain teams have responded with positive acknowledgments. Comprehensive details regarding their responses can be found in our repository at https://github.com/hzysvilla/Nurgle_Oakland24, and we will keep updating their feedback.

C.1. Vulnerable blockchains

NURGLE threatens 588 blockchains compatible with Ethereum and 153 blockchains compatible with Polkadot. We enumerate the blockchains in https://github.com/hzysvilla/Nurgle_Oakland24.

Appendix D. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

D.1. Summary

This paper presents a denial of service attack targeting the blockchains using the Merkle Patricia Trie (MPT) structure. The attack inserts new intermediate nodes to force the network to use more resources to keep the network state. The attack directly applies to popular blockchains such as Ethereum.

D.2. Scientific Contributions

- Identifies an Impactful Vulnerability

D.3. Reasons for Acceptance

1) The paper finds a novel denial service attack on popular blockchains.
2) The attack is tested on the testnet and acknowledged by the blockchain labs.
3) The paper demonstrates the attack is viable.
4) The paper is well-organized and easy to follow.