Published: 11 September 2024

Citation in BibTeX format

ISSTA '24: 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis
September 16 - 20, 2024
Vienna, Austria

Conference Sponsors:
SIGSOFT

RESEARCH-ARTICLE

Following the “Thread”: Toward Finding Manipulatable Bottlenecks in Blockchain Clients

SHUOHAN WU, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

ZIHAO LI, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

HAO ZHOU, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

XIAPU LUO, The Hong Kong Polytechnic University, Hong Kong, Hong Kong, Hong Kong

JIANFENG LI, Xi'an Jiaotong University, Xi'an, Shaanxi, China

HAOYU WANG, Huazhong University of Science and Technology, Wuhan, Hubei, China

Open Access Support provided by:

Xi'an Jiaotong University

The Hong Kong Polytechnic University

Huazhong University of Science and Technology

Following the "Thread": Toward Finding Manipulatable Bottlenecks in Blockchain Clients

Shuohan Wu
The Hong Kong Polytechnic University
Hong Kong, China
csswu@comp.polyu.edu.hk

Zihao Li
The Hong Kong Polytechnic University
Hong Kong, China
cszhli@comp.polyu.edu.hk

Hao Zhou
The Hong Kong Polytechnic University
Hong Kong, China
cshaoz@comp.polyu.edu.hk

Xiapu Luo*
The Hong Kong Polytechnic University
Hong Kong, China
csxluo@comp.polyu.edu.hk

Jianfeng Li
Xi'an Jiaotong University
Xi'an, China
jfli.xjtu@gmail.com

Haoyu Wang
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract

Blockchain clients are the fundamental element of the blockchain network, each keeping a copy of the blockchain's ledger. They play a crucial role in ensuring the network's decentralization, integrity, and stability. As complex software systems, blockchain clients are not exempt from bottlenecks. Some bottlenecks create new attack surfaces, where attackers deliberately overload these weak points to congest client's execution, thereby causing denial of service (DoS). We call them manipulatable bottlenecks. Existing research primarily focuses on a few such bottlenecks, and heavily relies on manual analysis. To the best of our knowledge, there has not been any study proposing a systematic approach to identify manipulatable bottlenecks in blockchain clients.

To bridge the gap, this paper delves into the primary causes of bottlenecks in software, and develops a novel tool named ThreadNeck to monitor the symptoms that signal these issues during client runtime. ThreadNeck models the clients as a number of threads, delineating their inter-relationship to accurately characterize the client's behavior. Building on this, we can identify the suspicious bottlenecks and determine if they could be exploited by external attackers. After applying ThreadNeck to four mainstream clients developed in different programming languages, we totally discover 13 manipulatable bottlenecks, six of which are previously unknown.

CCS Concepts

• **Security and privacy → Denial-of-service attacks**.

Keywords

Blockchain, Bottleneck, DoS Attack

*The corresponding author.

ACM Reference Format:
Shuohan Wu, Zihao Li, Hao Zhou, Xiapu Luo, Jianfeng Li, and Haoyu Wang. 2024. Following the "Thread": Toward Finding Manipulatable Bottlenecks in Blockchain Clients. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650212.3680372

1 Introduction

Blockchain technology has garnered significant attention in recent years, with its applications spanning finance, supply chain, IoT, and insurance sectors [41, 90]. According to CoinMarketCap, the total global market capitalization of mainstream blockchains is estimated at US 1.09 trillion in 2023 [2]. At the core of blockchain network are its clients — software entities that maintain a complete record of the blockchain, crucial for ensuring the network's decentralization [29]. Geth [14] is a notable example. However, like most complex software systems, blockchain clients are prone to bottlenecks. Certain bottlenecks present attack surfaces, where attackers can craft malicious input, deliberately overloading these weak points to congest client's execution and cause denial of service (DoS). When the number of active clients decreases, the blockchain network can be at risk. For instance, in Proof-of-Work (PoW), it heightens the risk of 51% attacks [74], while in Proof-of-Stake (PoS), it could disrupt the consensus process [74]. Moreover, since clients serve as backends for DApps[27], attacks on clients directly threaten the security of DApp ecosystem. For example, a DoS attack targeting client for a crypto wallet, could disrupt services for all wallet users. In this paper, we refer to these bottlenecks that could lead to DoS insecurity as manipulable bottlenecks.

Some manipulable bottlenecks in blockchain clients have already been identified. The Shanghai DoS attacks [23, 38] in 2016, for instance, exploited the EXTCODESIZE opcode, causing a surge in disk IO that makes clients crash. NURGLE [59] aggravates the consumed resources during blockchain state modification by manipulating the MPT structure of state storage. Further, research by Greene et al. [57] and Heo et al. [60] proved that some clients are vulnerable to transaction flooding attacks. However, to the best of our knowledge, there is a notable lack of a systematic approach to detect potential manipulable bottlenecks across blockchain clients. Most previous studies [48–50, 55] design benchmarks to intensify the load on specific parts until a bottleneck is observed. However, such a focused approach can only trigger a very constrained set of bottlenecks. Chen et al. [37] divide the client into three layers, evaluating the performance of each. This approach, though, offers a relatively coarse-grained view that might bypass bottlenecks within each layer. Efforts to instrument the client's source code for measuring latency across modules [76, 88, 89] also have limitations. Obtaining detailed latency insights requires extensive instrumentation, which in turn introduces performance biases, such as disk IO by logging. Moreover, given the sheer number of blockchain clients, instrumenting each client is not practical. Toyoda et al. [77] leverage traditional profiling tools to identify the most time-consuming functions. Still, longer execution times don't necessarily indicate a bottleneck (e.g., encryption functions naturally take longer due to complex hash operations). Moreover, given that a function can be invoked in various contexts, this approach cannot understand its actual performance impact.

To fill in the gap, in this paper, we propose a general-purpose tool, named ThreadNeck, to systematically identify potential manipulable bottlenecks in blockchain clients. ThreadNeck primarily operates by observing the runtime of clients, pinpointing behaviors that exhibit symptoms likely to cause bottlenecks. However, it is non-trivial to develop and design ThreadNeck. We face the following technical challenges.

C1: Characterizing Client's Execution. To effectively understand the client's execution and thereby identify the manipulatable bottlenecks, we need to accurately characterize each behavior within the client and their inter-relationships. However, compared to traditional software, blockchain clients present a higher degree of complexity and diversity, featuring a multi-layered architecture that includes network communication, data storage, consensus mechanisms, and smart contract execution engine, and more. Each layer introduces its unique complexities and requires intricate interaction with other layers. Given this, finding the appropriate granularity to depict the client's behavior poses a challenge. As previously discussed, manually dividing the client into several layers might overlook bottlenecks within each layer. Conversely, a function-level approach cannot grasp the context of calls and account for competition between multiple threads.

C2: Detecting Bottlenecks from Norms. For comprehensive identification, it is not practical to design a benchmark that forces a client to exhibit all bottlenecks. This is due to two primary reasons. First, unlike traditional software that can generate numerous unusual inputs through methods like fuzzing [40], producing diverse inputs for blockchain clients is difficult, not only because the inputs are varied and structurally complex, but it also needs to comply with consensus process and cryptographic standards. Second, given that clients are stateful and interact with various types of clients within the network, enumerating all execution scenarios is unfeasible. A more feasible approach is to observe the client's behavior during its regular execution, hunting for potential areas that might be exploited to cause bottlenecks. However, this method presents its challenges due to the absence of anomalous performance indicators. For example, when performance metrics like memory and CPU usage remain normal with regular operations, it becomes difficult to discern the subtle signs of a bottleneck.

C3: Crossing Client Diversities. The diversity of blockchain clients inherently means that each may have its distinct bottlenecks. Given this, our system is designed to be compatible with most mainstream clients, aiming to consistently identify the bottlenecks across such a varied landscape. However, this task is non-trivial because these clients are not only written in different programming languages, but also exhibit different designs. Such diversity makes it hard to provide a generalized solution.

To address **C1**, given that threads are the basic units of program execution, we model the client as a number of threads, and characterize the client's behavior by monitoring the lifecycles of these threads. Furthermore, we build a thread wait&wakeup graph for the client to illustrate the inter-relationships between threads. To address **C2**, we propose two primary root causes of bottlenecks in software systems, by examining existing research on performance diagnostics and optimization. During client execution, we observe specific symptoms that signal these issues to detect potential bottlenecks. Subsequently, we develop Proof-of-Concept (PoC) tests to verify their exploitability. To address **C3**, we compare thread models across different programming languages, proposing tailored implementation strategies for each.

We applied ThreadNeck to four popular blockchain clients for analysis. In total, we uncover 13 manipulatable bottlenecks, through which attackers could craft malicious input to DoS the client or specific modules, posing severe security threats. Remarkably, six of these bottlenecks are previously unknown. We have reported these findings to the respective developers, and at the time of writing, all of them have been confirmed.

Our major contributions are summarized as follows:

- We take the first step to systematically identify the manipulatable bottlenecks in blockchain clients.
- We design and implement a tool named ThreadNeck, which accurately characterizes blockchain client's execution to facilitate the identification of bottlenecks.
- We apply ThreadNeck to four mainstream blockchain clients. ThreadNeck totally finds 13 manipulatable bottlenecks, which can be exploited to cause severe security problems.

2 Background

2.1 Blockchain Client Architecture

A blockchain client is software, which maintains a record of the blockchain and communicates with other clients to verify and disseminate transactions and blocks. Geth (go-ethereum) is the most representative client, with recent statistics showing 3,000 nodes running it in the Ethereum network [5]. Figure 1 shows the Geth's architecture. We use Geth as an example, noting that other clients like Parity and Besu share a similar high-level architecture [28], as they need to conform to the Ethereum protocol [39]. Geth comprises the following five key components [35, 78]:

– **Network Layer**: manage communication with the underlying P2P network, exchanging messages with neighbor nodes.

– **Storage Layer**: store the state of the Ethereum network, maintaining all account balances, smart contracts [87], and the current state of these contracts in an underlying database.

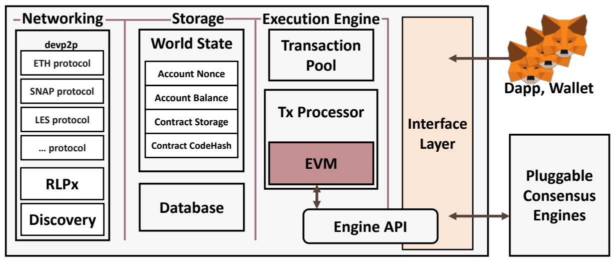


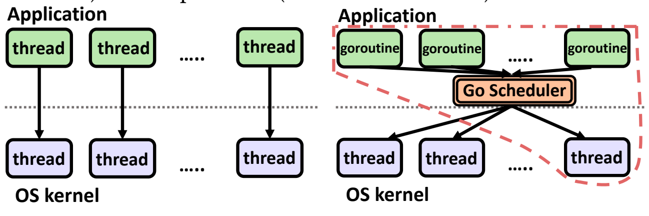
Figure 1: Geth architecture.

– **Execution Engine**: execute transactions and run the smart contracts. It comprises a transaction pool [68] for managing pending transactions and an EVM (Ethereum Virtual Machine) that executes contract's code, and computes blockchain's word state [83].
– **Interface Layer**: offer RPC, HTTP, and other interfaces, enabling external applications to interact with the client, e.g., sending transactions, and querying blockchain data.
– **Consensus Engine**: enforce rules that describe what the client should do to reach consensus about the broadcasted transactions. It also deals with the generation and verification of blocks. Note that, under the PoS, the Consensus Engine has been separated and interacts with the client through the Interface Layer.

When the user sends a transaction through DApp wallet to interact with a smart contract, the wallet first forwards this transaction as an RPC request to its connected blockchain client (i.e., Interface Layer), which then propagates it across the network using specific protocols (i.e., Network Layer). Clients capable of mining, upon receiving this transaction, add it to a new block. This newly created block is then broadcast throughout the network. Other clients in the network, upon receiving this block, use the consensus engine to validate it, and execute transactions within the block in EVM (i.e., Execution Engine). This updates the state info (such as the user's balance) stored locally on the client (i.e., Storage Layer), thereby recording the user's transaction on the distributed network.

2.2 Thread State

A thread is a path of execution in a program and can be in various states [10]. When actively executing on a CPU, it is in the **running** state. Conversely, a thread is **runnable** when it is ready for execution but waiting for the availability of the CPU. A thread enters the **blocked** state when it sleeps, or waits for a condition to be met, often due to synchronization primitives (e.g., locks, conditional variables) or I/O operations (disk and network IO).



(a) Native-threaded languages. (b) Golang.

Figure 2: Different languages' threading model.

2.3 Thread Model

Most prevalent programming languages, such as Java, C/C++, Rust, use native threads provided by the operating system (see Figure 2a). That is each thread within an application (e.g., a Java thread), corresponds to a separate OS thread.

Contrastingly, Golang adopts a different mechanism by implementing its lightweight thread, named goroutine. These goroutines are scheduled by the Golang scheduler to run on OS threads (see Figure 2b). This design effectively reduces the context-switching overhead, thereby supporting high concurrency. It also implies that the same goroutine can be executed by different OS threads.

3 Motivation

The goal of our study is to pinpoint the potential manipulable bottlenecks in blockchain clients. Bottleneck refers to a weak point in the software process that limits the application's efficiency. In this paper, we define a **manipulatable bottleneck** as certain bottleneck that can be influenced by external attackers, who deliberately overload these weak points to congest and restrict the client's (or its modules) execution, consequently making the client DoS.

3.1 Motivating Example

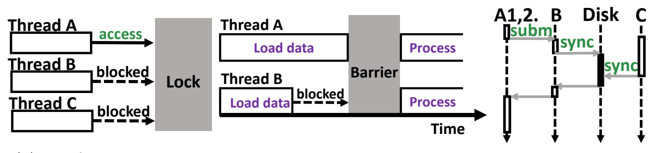
We use an example to explain how manipulable bottlenecks can be exploited. Disk read/write operations, a common performance bottleneck in software, were exploited in the Shanghai attack on Ethereum clients. In this attack, the attackers first deploy a contract (code provided below), which uses inline assembly statements to repeatedly call the `extcodesize` opcode —an instruction to retrieve the size of a contract's code. The attackers then sent a transaction to this contract. When the blockchain client processes this transaction, it is compelled to repeatedly search for the contract on disk, thereby inducing heavy IO operations that can lead to a DoS attack. This type of attack significantly affects all full-node clients in the blockchain network, as these nodes are tasked with downloading and executing every transaction within the block [26].

```
contract ExpensiveContract {
    function callExtcodeSize(address target) public
        view returns (uint256 size) {
       // Iteratively generates numerous EXTCODESIZE
        for(uint i = 0; i < 10000; i++) {
            assembly {
                size := extcodesize(target)
            }}}}
```

3.2 Root Causes of Bottlenecks

Upon survey of existing research on performance diagnostics and optimization [31, 69, 71, 91] and real-world DoS attack cases in blockchain clients [66, 67, 75, 81], we observe that there are mainly two causes of bottlenecks:

(1) Inefficient Synchronization: Synchronization is a mechanism in software used to ensure that threads execute in a specific order [25]. Inefficient synchronization refers to the improper use of this mechanism, resulting in some threads being unduly suspended, thereby affecting program performance. External attackers can exploit the weaknesses of inefficient synchronization by amplifying it, intensifying the synchronization issue and consequently causing other threads to wait excessively. Existing research [91] identifies three primary causes for these inefficiencies:



(a) Lock contention. (b) Load imbalance. (c) Blocking IO.

Figure 3: Three examples of inefficient synchronization.

• Lock contention. Excessive lock contention is a major inefficiency factor, causing threads to wait unnecessarily long for resource access. As shown in Figure 3a, attackers can increase the load on Thread A, thereby prolonging lock holding and forcing other threads into extended waiting.
• Load imbalance. In applications that involve phase-based parallel processing, load imbalance within the same phase can also lead to inefficiency. In Figure 3b, threads A and B execute a task in parallel, divided into two phases: "load" and "process". To enable the exchange of necessary data between threads A and B before the "process" phase begins, a barrier (i.e., a synchronization primitive) is used, to ensure both threads complete the "load" phase before moving on. Similarly, an attacker can increase the load on Thread A, prolonging its "load" phase, which consequently delays the "process" phase of others.
• Blocking IO. Blocking IO is another factor, as it can impede the threads that depend on its completion. As shown in Figure 3c, Thread B receives requests from A1 and A2 to perform blocking writes to the Disk. An attacker could issue a complex request to A1, triggering heavy IO operations on the Disk by B, thereby causing prolonged waiting for A2 (until the IO completion). Moreover, Thread C, which also needs Disk read and write access, becomes stalled since the Disk is preoccupied with B's extensive write operations. Note that, we consider the Disk as a pseudo thread. This simplifies our understanding of interactions between application threads and the Disk, categorizing them as a form of thread synchronization issue. Such a view also facilitates the design of a consistent method in our subsequent sections to identify such bottlenecks.
(2) Resource Exhaustion: Besides exploiting inefficient synchronization, external attackers could also trigger a code section that extensively uses system resources, such as CPU and memory, causing other threads to be delayed due to insufficient system resources.

To identify potential bottlenecks arising from **(1)Inefficient Synchronization**, we need to understand the interactions of threads within blockchain clients. Therefore, we model the client as a number of threads, and build a thread wait&wake graph. Using this graph, we can analyze the wake-up patterns between threads and track synchronization events the thread waits for, thus pinpointing areas prone to synchronization issues. To identify the bottlenecks arising from **(2)Resource Exhaustion**, we measure the resource consumption along the execution paths of each thread, pinpointing the paths most likely to exhaust system resources.

4 Overview of ThreadNeck

Figure 4 illustrates the workflow of our approach ThreadNeck, which is divided into two main components. The first component (detailed in §5) profiles the blockchain client at runtime. It records thread wake-up events and other relevant information, subsequently constructing a thread wait&wake graph. This graph effectively visualizes the inter-relationships between threads, thereby characterizing client's execution. The second component (detailed in §6) analyzes the built graph to detect manipulatable bottlenecks. To identify bottlenecks during client's regular execution, we monitor the symptoms that signal two types of bottlenecks we proposed in §3. This helps us identify suspicious threads that might lead to bottlenecks. For each identified thread, based on its call stack, we map it to its corresponding test case in the source code. Leveraging the template tests provided by developers, we then develop and run PoC tests to assess whether these threads could be exploited by external attackers to trigger a DoS attack.

5 Building Wait&Wake Graph

5.1 Thread Wait&Wake Graph

A thread wait&wake graph is a directed graph that captures the wait-and-wake-up relationships between threads within program's execution. As shown in Figure 5, each node in the graph represents a thread. Each node also holds the thread's call stack, which records the thread's execution path and can be considered as a specific behavior of the client. An edge from node A to node B indicates that thread A sometimes wakes up thread B. Alternatively, this also suggests that thread B sometimes waits for A. Consistent with §3, we also model both the NIC and Disk as pseudo threads in the graph. The inter-relationships of threads captured in this graph help us identify potential bottlenecks caused by inefficient synchronization, as explored in subsequent sections (§6.1). Moreover, to identify bottlenecks resulting from resource exhaustion (§6.2), we also record the resource usage along thread's execution path.

Blockchain clients are developed using various programming languages (see Table 1). As discussed in §2.3, these languages employ different thread models. Therefore, we implement tailored solutions for them, ensuring the wide applicability of ThreadNeck across a variety of clients available in the market.

5.2 Native-threaded Languages

Build wait&wake graph. For native-threaded languages that use threads provided by the OS, we build ThreadNeck based on wPerf [91]. wPerf hooks the kernel function `try_to_wake_up` using perf [16], which changes a thread's state from blocked to runnable. This function is typically invoked when conditions set by synchronization primitives are met (e,g, the release of a lock), or when receiving certain signals (e.g., SIGALRM) [19], or upon the I/O completions. By hooking this function, we can capture both the thread to be woken up, and the invoking thread. This allows us to establish the wait-and-wake-up relationships between threads. To include the NIC and Disk as pseudo threads, we also intercept the soft IRQ functions, which are triggered when an I/O completes to wake up the blocked thread (through `try_to_wake_up`). By doing so, we can know the type of hardware device (NIC or Disk) that is waking up the thread. To accurately capture thread's call stack (i.e., its execution path), we hook the `__switch_to` function, which gets invoked when a thread is switched out from the CPU.

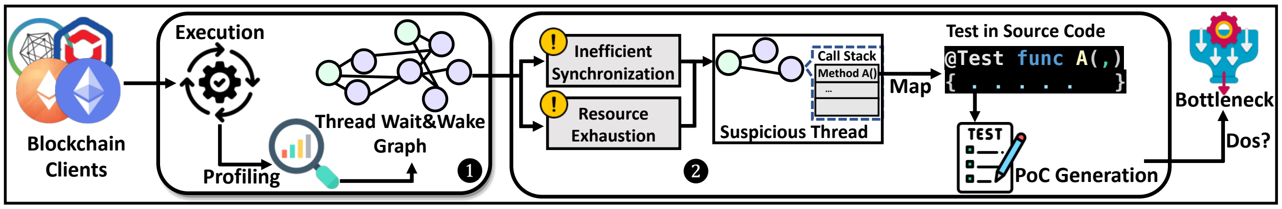


Figure 4: Workflow of ThreadNeck.

By collecting profiling data at the kernel level, we provide a consistent solution adaptable to clients developed in different native-threaded languages.

Handle thread reuse. Thread reuse is a common practice employed in modern software systems to minimize the overhead of thread creation and destruction, which is not considered by wPerf [91]. In some blockchain clients, we observe that, instead of creating an individual thread pool for a specific task, they employ global or shared thread pools that handle tasks from different modules. This practice further complicates our analysis, because in our wait&wake graph, we make the assumption that each thread is dedicated to a single task (i.e., an execution path). Therefore we can identify, through the thread, the specific task causing the bottleneck. The code below illustrates such an example. The thread A (from the thread pool), after completing Task1, is reused for Task2. If Task2 causes a bottleneck, resulting in long waiting for Thread B, our graph cannot accurately pinpoint the actual execution path (i.e., the code section) responsible for the bottleneck.

```
void Task1_ThreadA() { //Thread A executes Task1.
  //benign code }
void Task2_ThreadA_Reuse(){//Thread A is reused.
  pthread_mutex_lock(&lock);
  bottleneck();
  pthread_mutex_unlock(&lock);}
void ThreadB()  {
  pthread_mutex_lock(&lock);//B waits for Task2's lock.
    ......}
```

To address this, we break down threads in the thread pool into more fine-grained pseudo threads, based on the distinct tasks they execute. This is exemplified in Figure 6, where Thread C is further split into C-1 and C-2 (i.e., pseudo threads) in the wait&wake graph, as they are responsible for different task executions. To differentiate these tasks and split the threads, we try to identify the exact moments before the thread is assigned a new task (i.e., be reused). Our observation indicates that, when the threads are waiting for task assignments, they are typically in a blocked state, blocked by some synchronization mechanisms in the thread pool. For example, Figure 7 shows a Java thread blocked by thread pool’s queue while idle. Therefore, when a thread is switched out of the CPU (recall that we hook the kernel function `__switch_to`), we examine its call stack to determine if it is idling in the thread pool. If true, we record this idle time, and differentiate tasks before and after this time point. In Figure 6, from the call stack of Thread C obtained

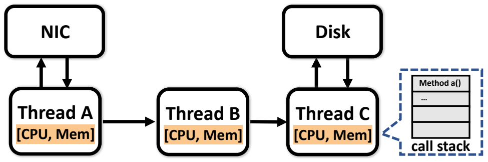


Figure 5: Thread wait&wake graph.

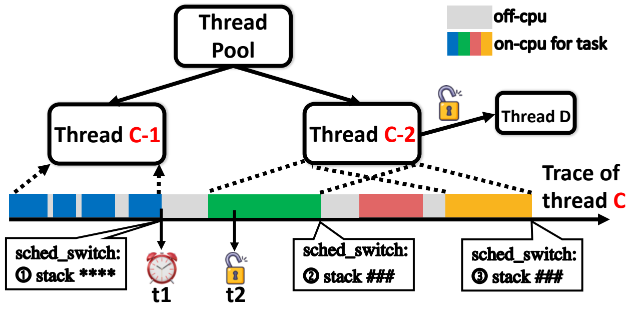


Figure 6: Thread reusing.

at `t1`, we can determine it is idling in the pool, indicating the task prior to `t1` on Thread `C` is complete, and any task after is a new one. Consequently, we split Thread `C` into distinct pseudo threads for each task (how we merge similar threads is detailed in §5.4). Assume that, at time `t2`, we detect Thread `C` wakes up Thread `D`, this wake-up relationship will be added to Thread `C-2`. Note that, our current focus primarily addresses thread reuse from the thread pools, as this is the most common reuse scenario.

Measure resource usage. For each thread in the thread wait&wake graph, we also account for their system resource usage to detect potential bottlenecks caused by Resource Exhaustion (see §3). We primarily focus on two critical system resources: memory and CPU. For memory measurement, it is challenging to measure the precise memory usage of each thread. Because for most languages, their threads use shared memory. This means that no memory is strictly owned by any particular thread, thus making it hard to determine what thread each object (e.g., data structure, variable) belongs to [8]. To tackle this issue, we employ useful profiling tools (e.g., async-profiler [21]) to sample the memory allocations in each method. Therefore, we can estimate a thread’s memory consumption based on the methods executed in its call stack. To assess CPU usage, similarly, we gather function execution time from the profiling tools. We then estimate the CPU time consumed along a thread’s execution path based on its call stack.

5.3 Golang

Build wait&wake graph. As discussed in §2.3, Golang implements its own lightweight threads (i.e., goroutines). To construct the goroutine wait&wake graph, we need to develop a tailored approach, as directly observing threads at the OS level is not feasible in this context. This is due to Golang’s thread model, where a single OS thread can run multiple goroutines. In fact, to maximize CPU utilization, the Golang runtime often keeps the OS thread in **running** state when scheduling different goroutines on it, minimizing its chances of being swapped out from the CPU [22].

```
"worker-thread-10"
at jdk.internal.misc.Unsafe.park
......
at java.util.concurrent.ForkJoinPool.managedBlock
at locks.AbstractQueuedSynchronizer.await
|at ScheduledThreadPoolExecutor$DelayedWorkQueue.take
     |
|at ThreadPoolExecutor.getTask|
|at java.util.concurrent.ThreadPoolExecutor.runWorker
     |
at java.lang.Thread.run
```

Figure 7: Call stack of idle threads in thread pool.

Luckily, after inspecting the Golang's source code, we find that all goroutine wake-ups in the Golang are achieved through the `goready` function. As shown in Figure 8, we instrument the `goready` function, recording the current goroutine (Line 2), which triggers the wake-up, and the goroutine that is waiting to be woken up (Line 3). It's important to point out that, while there are schedulers within the Golang runtime that are responsible for goroutine scheduling [24], they do not directly participate in waking up threads. To illustrate, when goroutine A wakes up B by releasing a lock, the actual invoker of `goready` is A, not the scheduler.

Measure resource usage. We leverage the tools trace [20] and pprof [18] to account for resources (i.e., CPU, memory) consumed by each goroutine and method in the target client.

5.4 Merge Similar Threads

As a large-scale software system, blockchain clients run numerous threads during their execution. This results in a massive thread wait&wake graph, complicating our effort to identify manipulatable bottlenecks in §6. To this end, we aim to reduce the graph's size by merging similar threads. The rationale behind this approach is that many threads are used to perform same tasks. For example, in the network module, to improve the throughput, blockchain clients employ multiple threads to concurrently handle network requests.

We identify similar threads based on their call stacks. Considering the vast number of threads and their extensive call stacks, to improve computation efficiency, we represent each thread as a sparse vector and compute the cosine similarity between them. More specifically, we first develop a vocabulary by cataloging unique methods from the call stacks of all threads, with each method assigned a distinct position. Then, for each thread, we merge its call stacks captured during profiling, into a sparse vector. In this vector, a method's position is marked as 1 if present in the thread's call stack; otherwise, it remains 0. Using these vectors, we can compute the cosine similarity between threads. This approach not only allows for rapid comparison of task similarities executed by threads, but also accounts for potential imperfections inherent in sampling (wPerf in §5.2 is built on perf, a sampling-based tool). The cosine similarity threshold is made a configurable parameter. We discuss its impact and provide suggested value in §7.5.

6 Identify the Bottlenecks

After building the thread wait&wake graph, we try to identify the potential manipulatable bottleneck, targeting Inefficient Synchronization and Resource Exhaustion:

```
1 func goready(gp *g, traceskip int) {
2   gpFrom := getg()
3   gpTo := gp
4   //...record Goroutine wake-up
5   systemstack(func() {
6     ready(gp, traceskip, true)
7   })}
```

Figure 8: Golang source code instrumentation.

6.1 Inefficient Synchronization

We observe that the primary indicator of inefficient synchronization is the presence of cyclic waiting among threads. Recalling the three primary causes for the inefficiency synchronization discussed in §3, in all these scenarios, threads exhibit cyclic waiting due to dependencies in their execution sequence. For instance, in Figure 3a, Threads B and C must wait for Thread A when it holds a lock, and conversely, Threads A and C wait for Thread B when it does. In Figure 3b, Threads A and B encounter mutual waiting due to varying completion times in the "Load" phase under different circumstances. Similarly, in Figure 3c, thread B waits for the Disk during a blocking write, and when the Disk is idle, it awaits data from thread B.

Hence, to identify potential bottlenecks caused by inefficient synchronization, our focus is on detecting **cycles** in the wait&wake graph. Each cycle represents a cyclic wait-for relationship for threads. Our focus is on threads woken up by the NIC (which represents the external input). For example, in Figure 9, there is a cycle between `Eth-importBlock` thread and Disk. The reachable path from NIC to `Eth-importBlock` suggests that external inputs influence this thread, potentially causing extensive blocking I/O operations on Disk and impacting the execution of other modules.

Exclude Thread Pool Cycles. In practice, we find that many cycles in the graph arise from the interplay between thread pools and their worker threads. This dynamic resembles a producer-consumer relationship, when the task queue is empty, the worker threads wait for the thread pool to submit tasks. Conversely, when the queue is full, the thread pool's attempt to submit new tasks is blocked, waiting for worker threads to retrieve tasks from the queue. To simplify our analysis, we comment out all cycles involving thread pools and their worker threads (In Figure 9, cycles between Eventloops and their worker threads are represented by dashed lines).

However, it's important to note that, improperly configured thread pools, such as having too limited worker threads and queue sizes, could also be bottleneck. Therefore, for thread pools identified in the graph, we perform a manual code inspection to check their configurations. This helps us determine if attackers could deplete the thread pool by flooding it with numerous specific tasks.

Write PoC Tests. To validate whether these instances of inefficient synchronization can be exploited, we design and write PoC tests. Crafting these tests presents its own challenges. On one hand, blockchain clients are often tightly coupled systems with interdependent modules; on the other hand, it also requires deep familiarity with the client's source code, such as constructing objects and using API. Fortunately, we find almost all clients provide rich unit tests for their modules in the source code, which we use as templates for rapid PoC tests implementation. To efficiently locate unit tests corresponding to each observed cycle, we search the source code using class and method names based on the call stacks of thread in the cycle. These unit tests are typically located in files named with the class name followed by "Test" (see Figure 9).

When writing PoC tests, if it is a known issue, such as previously documented attacks or disclosed vulnerabilities, we employ strategies that have proven effective in similar scenarios. Moreover, we define some strategies to refine PoC tests, aiming to increase the load on threads involved in the cycle. This involves methods such as increasing the number of transactions, enhancing transaction complexity, expanding block size, and amplifying the volume of concurrent requests. For unknown issues, we integrate the examination of call stacks (i.e., the thread execution paths) and the corresponding source code to identify the causes of bottlenecks. Specifically, we analyze which functions contribute to cyclic waiting among threads. We then assess whether these functions could be influenced by external inputs, such as parameters derived from external messages. This allows us to determine whether it’s feasible to manipulate external inputs to intensify these issues. We showcase our PoC development with a practical example in §6.3.

6.2 Resource Exhaustion

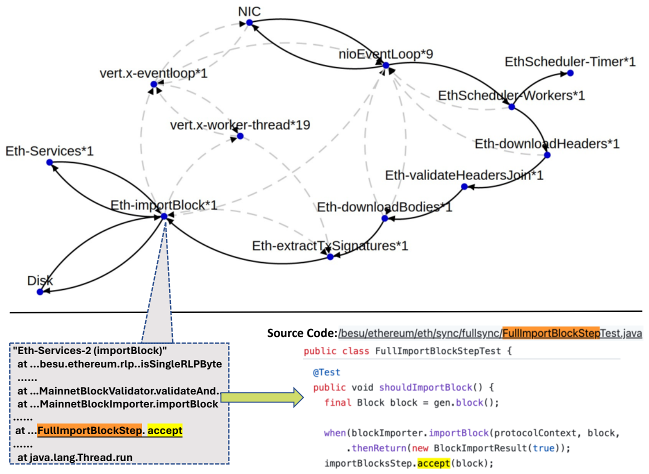
Identifying code sections that might drain system resources presents a challenge. Static analysis cannot observe the actual resource usage. Moreover, path-insensitive static analysis generates too many false positives, path-sensitive analysis faces scalability issues with large codebases like those of blockchain clients [53]. Traditional dynamic analysis [58], which monitors each function’s resource usage, is impractical. Blockchain clients have numerous functions, which require extensive manual effort for analysis.

To address this, we assess the resource usage of each thread to identify those most likely to exhaust system resources. Our primary focus is on two types of threads: the first type includes those with long execution times, or significant memory allocations, which attackers could exploit by causing the client to concurrently execute these threads, leading to resource exhaustion. The second type includes threads with high variability in execution time or memory allocation, suggesting that their performance may be greatly influenced by external inputs. Attackers may craft malicious inputs to increase resource consumption in these cases. We use standard deviation as a metric to gauge the extent of this variability. For both thread types, we set configurable thresholds for analysts. Their impacts are also discussed in §7.5.

Write PoC Tests. We construct PoC tests to assess the exploitability of identified threads, based on the developer-provided unit tests. We analyze functions within the thread call stacks that cause significant resource consumption or display exhibit variability in resource use. For these functions, we examine whether their parameters can be manipulated by external inputs. In designing the PoCs, similarly, we incorporate insights from known issues alongside manual analysis for those issues yet to be identified. Our goal is to amplify the workload on these critical functions to create extreme scenarios.

6.3 Demo Example

Figure 9 presents an example from part of Besu’s thread-wait&wake graph. The graph shows Besu’s workflow, which is primarily a pipeline. For example, it first downloads the block header, validates it, and then proceeds to download the block body. After verifying



```
"Eth-Services-2 (importBlock)"
 at ...besu.ethereum.rlp..isSingleRLPByte
 ......
 at ...MainnetBlockValidator.validateAnd..
 at ...MainnetBlockImporter.importBlock
......
at ...FullImportBlockStep. accept
......
 at java.lang.Thread.run
```

Source Code:/besu/ethereum/eth/sync/fullsync/FullImportBlockStepTest.java

```
public class FullImportBlockStepTest {

  @Test
  public void shouldImportBlock() {
    final Block block = gen.block();

    when(blockImporter.importBlock(protocolContext, block,
        .thenReturn(new BlockImportResult(true));
    importBlocksStep.accept(block);
```

Figure 9: A demo example of Besu’s full sync. where we first identify a cycle indicative of Inefficient Synchronization. Then we use the call stacks of threads within the cycle to find the corresponding test cases in the source code.

all the transaction signatures in the body, it finally imports the block into the chain. Within the graph, a notable cycle between the `Eth-importBlock` thread and Disk indicates a potential bottleneck caused by inefficient synchronization.

To validate its exploitability, we craft a PoC test for the `Eth-importBlock` thread. Specifically, by inspecting the thread’s call stack, we identify the task’s entry point as the `FullImportBlockStep.accept`. Thus, we locate the corresponding unit test provided by developers in the source code by matching the class name and method name (in the bottom right of the figure). This unit test primarily generates a block and simulates block importation. To analyze the cause of the cyclic wait between `Eth-importBlock` and Disk, we delved further into its call stacks. We notice calls to the `SLoadOperation.execute()` and `SStoreOperation.execute()`, which are EVM opcodes for loading and storing data from storage, respectively. This observation suggests that updates to contract states in transactions are inducing Disk IO. To amplify this issue, we deploy a contract as shown in Figure 10. This contract adds elements to an array in its state, triggering extensive updates (using `SLoadOperation` for each `push`).

```
contract IOHeavy {
bytes32[] private storage;
    function heavyUpdate (uint256 num) public {
        for (uint i = 0; i < num; i++){
            storage.push(sth);
}}
```

Figure 10: Contract with numerous storage updates.

Following this, we modify the unit test, to include transactions that interact with this contract in the generated block. We monitor whether it causes the test to experience extended execution times, thereby validating the presence of a bottleneck.

7 Evaluation

We evaluate the performance and functionality of ThreadNeck by answering the following four research questions (RQs).

Table 1: Blockchain clients used in our experiment.

Client	Abbr.	Version	Language	LOC
Go-Ethereum [14]	Geth	1.11.5	Golang	250,303
Hyperledger Besu [4]	Besu	23.4.1	Java	342,997
Reth [11]	Reth	0.1.0-alpha.1	Rust	136,273
FISCO-BCOS [6]	BCOS	3.2.2	C++	172,928

"Abbr." stands for abbreviation for client, "LOC" stands for Line of Code.

RQ1: Can ThreadNeck effectively identify the manipulatable bottlenecks in the blockchain clients?
RQ2: Is ThreadNeck more effective in bottleneck identification compared to existing tools?
RQ3: How much overhead ThreadNeck introduces during its use?
RQ4: How does the configurable threshold in ThreadNeck impact the bottleneck detection?

7.1 Environment Setup

We apply ThreadNeck to four popular Ethereum clients in the experiment. We select Ethereum due to it is one of the most widely adopted blockchain platforms with substantial economic implications [12]. Table 1 lists details about them. Given that ThreadNeck observes client execution at runtime to detect bottlenecks, we synchronize each client with the mainnet, using real-world transactions on mainnet as input. This practice offers two advantages: firstly, the behavior on the mainnet is more diverse, and secondly, it's more likely to capture potential issues that occur in the real world. For BCOS, which has no publicly accessible network, we reuse the transactions from the Ethereum mainnet as it is EVM-compatible, Considering that clients can also be communicated with through JSON-RPC and GraphQL APIs, which are not present on the mainnet, we also manually simulate different RPC and GraphQL call patterns. We run each client for four days. All experiments are conducted on a 64-bit machine with 16 cores and 256GB memory.
Profiling. ThreadNeck profiles the blockchain clients based on sampling (both perf and pprof, on which it is built, are sampling-based). Running ThreadNeck for long time can result in an excessive volume of data, making both processing and analysis time-consuming. Therefore, inspired by [54], we adopt a sliding time-window profiling strategy. For each client, we perform profiling every two hours, with each session lasting 120 seconds. We set the sampling frequency at 4000Hz, according to [17], this frequency is adequate for gathering necessary data with minimal overhead. Higher frequency does not show great improvements in accuracy.
Compiler options. To obtain more complete threads' call stacks, we enable program's frame pointer for unwinding stack traces. For Besu, we run it with `XX:+PreserveFramePointer` option, which helps retain the frame pointer of the current method. Additionally, we also employ the perf-map-agent tool [1] to map Java code addresses to their corresponding method names. For Reth, we compile it with `force-frame-pointers=yes` flag. The frame pointer for BCOS and Geth is enabled by default.

7.2 RQ1. Effectiveness of ThreadNeck

Result. Table 2 presents the details of the manipulatable bottlenecks we identified in four blockchain clients. This includes seven known DoS bottlenecks and six newly discovered ones. We have reported our newly found bottlenecks to the developers. At the time of writing, all have been confirmed, with three of them being assigned a CVE ID.

In Figure 11, we illustrate three cases, demonstrating how ThreadNeck identified these bottlenecks. Each sub-figure shows part of the thread wait&wake graph from different clients, where we use solid lines to show the paths utilized by attackers to exploit bottlenecks. For Geth and Besu, we name most threads using the thread ID provided by perf and pprof. In cases of reused worker threads in thread pool, we manually check their call stacks to determine their thread names. For BCOS and Reth, where perf only provides numerical thread IDs, we also manually name the threads.

Figure 11a shows a part of Geth's behavior. ThreadNeck identified a cycle among IO-2(NIC), `Peer`, and `Downloader` threads. This cycle is formed because the `Downloader` requests the download of block headers and bodies, waking up the `Peer dispatcher`. The `dispatcher`, in turn, handles requests and then signals the `Peer readloop`, which writes the requests to the IO-2(NIC). When the response returns, the `Peer readloop` reads data from the IO-2(NIC) and wakes up the `Peer startProtocols`, which registers many handlers for handling messages. A handler then wakes up the `Downloader` to notify that the data has been received. This leads to a manipulatable bottleneck (ID ⑤), where a malicious master peer can significantly delay the sync process (i.e., the `Downloader` thread), by disrupting the client's sync operation each time it nears completion [75]. Additionally, we observe a cyclic wait between `legacypool loop` and `TxFetcher fetchTxs`. While we did not identify a bottleneck caused by inefficient synchronization in this cycle, a manual inspection revealed a logical bottleneck (ID ⑧). Specifically, an attacker can send invalid transactions or txs with high fees to evict normal transactions from the pool [67]. While this issue has been fixed in Geth, we found it remains unpatched in the BCOS client. Moreover, the graph also reveals a cyclic wait between `BlockChain insertChain` and Disk, stemming from the former's read-write operations on the trie (data storage for world state) while executing transactions in a block. An identified exploit (ID ①) is to deploy a smart contract that triggers numerous storage updates (as shown in Figure 10) and then invoke it via an `eth_call` RPC request (a transaction type that doesn't consume gas). Beyond cyclic waits, we also observe great variability in the execution time of the `BlockChain insertChain` thread. Further analysis links this issue to the complexity of the contracts it executes. Thus, an attacker could deploy a contract involving extensive hash computations (e.g., call `keccak256` in the contract) and use multiple `eth_call` to invoke it, leading to CPU resource exhaustion (ID ②).

In Figure 11b, ThreadNeck identifies a bottleneck as a cycle between the `batchPersistStorageData` thread and Rockdb (a k-v database). The former stores the downloaded blockchain world state into Rockdb, which then flushes this data to disk. Our investigation finds that when Rockdb encounters a busy exception due to conflicting writes, the `batchPersistStorageData` thread will hang, because there is no code logic to handle this exception and retry the operation (ID ⑨). Figure 11c displays part of Reth's wait&wake graph while processing certain RPC calls. Typically, upon receiving RPC requests from its lower network module (i.e., `revm_network` thread), Reth activates a dedicated thread to handle each request, a practice observed in other clients as well. In

Table 2: Manipulatable bottleneck and their exploits.

#ID	Bottlenecks	Known	Possible Exploitation of Bottlenecks (PoC)	Client
①	EVM ⇄ Disk	Re-confirm	Create numerous Ethereum accounts to increase the I/O operations on the trie.	Geth, Besu, Reth, BCOS
			Execute numerous `eth_call` requests simulating transactions with heavy SSTORE opcode to increase I/O on the trie.	
②	EVM ⊘ CPU	Re-confirm	Execute numerous `eth_call` requests that run a loop of hashing computation.	
			Flood client with transactions that contain a large number of cheap instructions yet high CPU consumption.	
③	EVM ⊘ Mem	Re-confirm	Execute numerous `eth_call` requests that allocate large memory.	
④	Debug Trace ⊘ Mem	Re-confirm	send numerous `debug_trace_` request to get the full trace of complex transactions.	
⑤	Downloader ⇄ NIC	Re-confirm	Manipulate malicious master peer to disrupt client's sync operation as it nears completion.	
⑥	p2p server ⇄ NIC	Re-confirm	Manipulate multiple peers to send extensive sync requests to the client, which in turn generate asymmetrically large sync responses (e.g., body).	
⑦	GetLogs ⇄ Disk	Re-confirm	Execute `eth_getLogs` request to retrieve logs over a massive block range.	
⑧	TxFetcher ⇄ TXPool	No	Submit invalid transactions to evict/pend executable transactions in txpool.	BCOS
⑨	Downloader ⇄ RocksDB	No	Submit multiple transactions that modify the same account, creating database write conflicts, which can make Downloader stuck at RocksDB due to the lack of a retry mechanism.	Besu
⑩	Vert.x ⇄ NIC	No	Establish many open connections with clients, since JVM will accept indiscriminately all incoming connections [13].	Besu
⑪	TxPool ⊘ Memory	No	Send numerous blob transactions which carry a large amount of blob data.	Besu
			Submit a large number of oversized transactions, exploiting the lack of size checks in transaction verification.	BCOS
			Send numerous transactions using several accounts, exploiting the unlimited transactions per account.	BCOS
⑫	Downloader ⊘ Mem	No	Reth's downloader parameters(e.g., buffer size) are constant, which risks crashing systems with limited memory.	Reth
⑬	Downloader ⇄ NIC	No	Manipulate peer to halt the download of a specific block, leveraging the fact reth downloads block ranges one by one.	Reth

[1] ⊘ denotes bottlenecks caused by resource exhaustion. A ⊘ B indicates thread A exhausts resource B.
[2] ⇄ denotes bottlenecks due to inefficient synchronization. A ⇄ B implies a cyclic waiting between threads A and B.
[3] In the "Known" category, "Re-confirm" indicates re-confirmation of previously known bottlenecks, "No" indicates newly discovered bottlenecks confirmed by developers.

Table 3: Mitigation strategies.

ID	Mitigation
①-⑦,⑩	Limit remote RPC calls and the number of concurrent RPC.
⑧	Detect latent overdraft transactions when replacing tx in the pool.
⑨	Add retry logic where exceptions are caught.
⑪	Incorporate detailed transaction checks in line with Geth's approach.
⑫	Adaptively adjust download parameters based on the available memory.
⑬	Implement logic for downloading blocks in batches in downloader.

the `revm_utils` thread, which handles `debug_traceTransaction` requests, we note that the `run_interpreter` function exhibits substantial memory usage variations. Analysis reveals that attackers can exploit this (ID ④) by submitting complex transactions through `debug_traceTransaction` requests. These requests generate a trace for each execution step of the instructions, resulting in the accumulation of extensive trace data in memory. The complexity of these transactions, due to their numerous instructions, can lead to memory exhaustion. Moreover, the graph also reveals a cycle involving `EthGetlogs`, `libmdbx`, and Disk. Attackers can exploit this (⑦) by using `eth_getlogs` requests to retrieve logs over an expansive block range. With the need to scan a vast number of blocks, the client must sequentially scan the entire database, causing significant Disk IO and blocking other disk-reliant modules.
Mitigation. In Table 3, we propose mitigation strategies to address the manipulatable bottlenecks we identified. For most of them, the exploitation is initiated through RPC requests, so a straightforward solution is limiting remote RPC connections. For newly discovered bottlenecks, our investigation reveals that they primarily result from coding oversights rather than inherent architectural flaws. Patching the affected code can effectively mitigate these issues.

7.3 RQ2. Overhead

We measure the overhead introduced by ThreadNeck on CPU and memory usage. To achieve this, we conducted ten separate samplings for system CPU and memory, with a frequency of once per second and each session lasting for 30 seconds. Table 4 reports the experiment result. We can find that, ThreadNeck introduces minimal overhead. On average, it leads to a 9% increase in CPU usage, while its impact on memory consumption is negligible.

7.4 RQ3. Comparison to Existing Tools

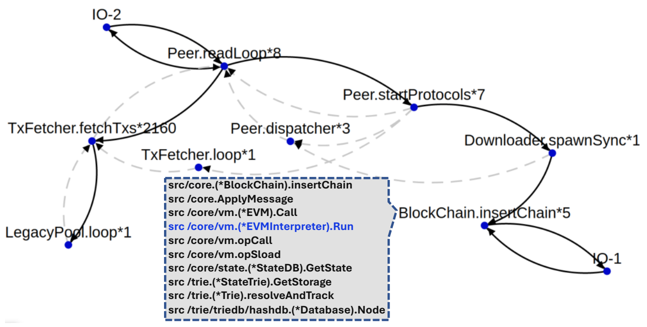
We compare ThreadNeck with the previously developed tool, wPerf [91], which identifies bottlenecks by pinpointing waiting events that are critical to throughput. Since wPerf currently supports only Java and C++, we apply it to Besu and BCOS.
Methodology. We run wPerf following our experimental setup, sampling for 120 seconds every two hours. wPerf introduces several user-configurable parameters, for which we use the default values. We manually reviewe all bottlenecks reported by wPerf to check if they match our identified manipulable bottlenecks, or if there are any new suspected bottlenecks.
Result. wPerf reported bottlenecks labeled ⑤⑥⑧. It did not discover any new suspected bottlenecks beyond those we had already identified. This is because, wPerf also searches for cycles of inter-thread waiting. wPerf's observation is that optimizing threads outside the cycle will not improve the throughput of threads within the cycle, hence considering these cycles as bottlenecks of throughput.

wPerf fails to recognize bottlenecks ⑦ ⑨, due to the thread reuse issue in Besu, where multiple modules share a thread pool. wPerf does not recognize thread reuse, leading to the masking of critical execution paths that contribute to bottlenecks. Regarding other bottlenecks missed by wPerf, e.g., ①-④, the main reason is that wPerf ignore the bottlenecks caused by resource exhaustion.

Table 4: Overhead of ThreadNeck.

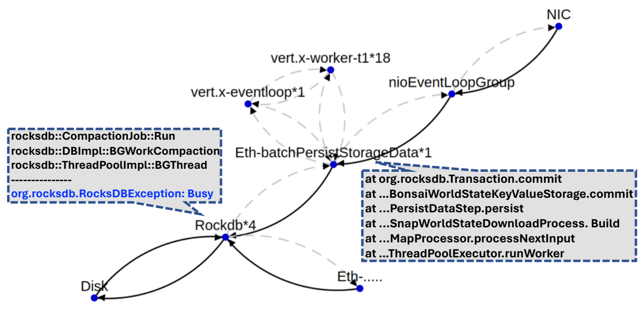
Client	CPU	CPU (+T)	Memory	Memory (+T)
Geth	597.7%	628.8%	14.76GB	14.62GB
Besu	326.1%	362.8%	23.12GB	23.33GB
Reth	170.1%	188.7%	50.6 GB	50.6GB
BCOS	16.3%	18.4%	0.39GB	0.40GB

(+T) indicates the client is running with ThreadNeck.



```
src/core.(*BlockChain).insertChain
src /core.ApplyMessage
src /core/vm.(*EVM).Call
src /core/vm.(*EVMInterpreter).Run
src /core/vm.opCall
src /core/vm.opSload
src /core/state.(*StateDB).GetState
src /trie.(*StateTrie).GetStorage
src /trie.(*Trie).resolveAndTrack
src /trie/triedb/hashdb.(*Database).Node
```

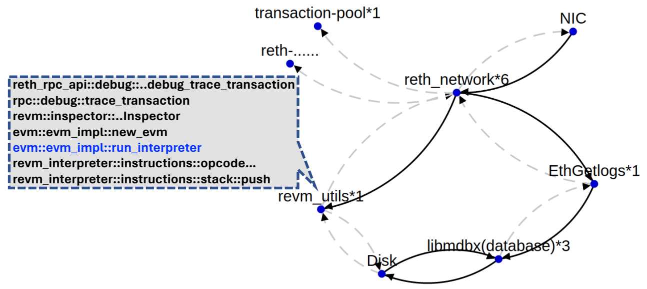
(a) Geth synchronizing new blocks.



```
rocksdb::CompactionJob::Run
rocksdb::DBImpl::BGWorkCompaction
rocksdb::ThreadPoolImpl::BGThread
---------------
org.rocksdb.RocksDBException: Busy
```

```
at org.rocksdb.Transaction.commit
at ...BonsaiWorldStateKeyValueStorage.commit
at ...PersistDataStep.persist
at ...SnapWorldStateDownloadProcess. Build
at ...MapProcessor.processNextInput
at ...ThreadPoolExecutor.runWorker
```

(b) Besu processing batch downloaded data.



```
reth_rpc_api::debug::..debug_trace_transaction
rpc::debug::trace_transaction
revm::inspector::..Inspector
evm::evm_impl::new_evm
evm::evm_impl::run_interpreter
revm_interpreter::instructions::opcode...
revm_interpreter::instructions::stack::push
```

(c) Reth processing RPC requests.

Figure 11: Case study for identifying bottlenecks.

7.5 RQ4. Effects of Thresholds

We use Besu as an example to investigate how thresholds affect our analysis. Besu is selected since it has the highest number of threads. We introduced two thresholds in this study: one for merging similar threads and the other for filtering threads prone to resource exhaustion. For the first threshold, Figure 12 illustrates the impact of varying this threshold on the number of edges and nodes in the thread wait&wake graph. A 100% threshold means that two threads are merged only if their similarity is absolute. We compare the wait&wake graphs generated with thresholds set at 50%, 60%, and 70%. The graph at 50% misses a bottleneck (TxFetch), while those at 60% and 70% report all identified bottlenecks. Therefore, for Besu, we use 60% as threshold. For Geth, we select 70% threshold. We set threshold 100%, for BCOS and Reth, considering their limited thread count. For the second threshold, we analyze the top 5, 10, 15, 20 threads in terms of resource consumption. Our findings suggest that a top 15 threshold is reasonable for most clients, setting it lower could result in missed detections of some bottlenecks.

8 Discussion

The method described in our paper is general and could be applied to many other software. This is because "Inefficient Synchronization" and "Resource Exhaustion" are common causes of bottlenecks in modern software, as revealed by our investigation. Our approach looks beyond specific behaviors exhibited during software execution to focus on the essential characteristics of bottlenecks.

We chose blockchain clients as our analysis target not only because of their critical importance to the web3 community, but also due to the blockchain community's commitment to open-source. The availability of open-source client code and rich unit tests greatly facilitates our analysis of the causes of bottlenecks in the application and the development of PoC tests. Additionally, despite their design differences, blockchain clients generally work in similar ways, allowing us to utilize experiences from other clients when designing PoC tests. For example, with bottleneck ID 6 in Table 2, this issue has previously led to a DoS attack in Geth. Consequently, we can use the strategy that was effective in Geth's DoS attack to develop PoC tests for Reth.

we acknowledge that our method may not be suitable for all types of software. For example, in multi-process software, thread in one process often use IPC (inter process communication) such as socket to interact with others, which our approach might not effectively address. Moreover, software with simple thread models (with almost no interaction between threads), making it hard to identify inefficient synchronization issue.

9 Threat To Validity

The main threat to external validity in our study is the reliance on manual effort for writing test cases. While manually designed test cases are of high quality, they can be time-consuming and less scalable. In future work, we plan to leverage large language models, like ChatGPT to generate test cases based on developer-provided templates. We also tend to integrate fuzzing techniques to increase the diversity of the tests.

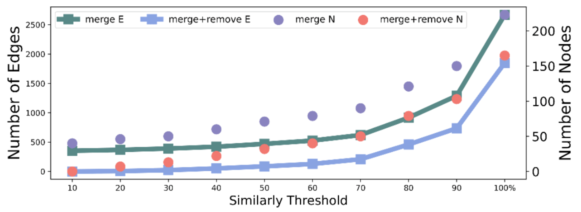


Figure 12: Edges and nodes in the graph at different thresholds. "Merge E" shows edges after merging similar threads; "Merge + Remove E" counts edges after merging similar threads and removing JVM threads.

Another threat to external validity is the limited number of blockchain clients included in our experiments. For instance, Nethereum, a popular client developed in CSharp, is not included. However, our study covers most mainstream clients and programming languages. Our future work will expand our experimental scope to incorporate more clients.

Moreover, some bottlenecks do not necessarily lead to a direct decrease in system performance. For instance, we find invalid transactions can evict normal transactions from the thread pool. These types of bottlenecks typically arise from logical issues rather than performance constraints. Detecting such bottlenecks relies on designing specific oracles, and cannot be straightforwardly inferred through thread behavior observation alone.

Lastly, during the profiling, the dynamic execution of clients didn't cover all code paths. This limitation was partly due to the presence of dead code (e.g., deprecated in newer versions), and partly because our execution scenarios were still limited. In the future, we aim to employ message fuzzers like [42] to trigger more code paths, which may help identify more potential bottlenecks.

10 Related Work

10.1 Bottleneck Analysis

On-CPU Analysis. Numerous general performance profilers have been implemented, through instrumentation [3, 7] or sampling [9, 15], to rank code based on its contribution to total execution time, identifying functions that consume most time. However, code that exhibits long running times does not necessarily represent a bottleneck. COZ [45] introduces a novel method, virtually accelerating some code execution to estimate the potential improvements from optimizing a certain code. However, its scalability is limited, especially in complex systems like Blockchain clients, due to their large codebases and the challenges in measuring throughput and latency. Other works diagnose bottlenecks by comparing execution traces of the same tasks in different contexts [51], different app versions [51], and configuration changes [62]. However, this approach relies on reproducing bottlenecks during dynamic testing, which can be triggered only under specific inputs or conditions.

Off-CPU Analysis. Some waiting events may also serve as the root cause of a bottleneck. Alam et al.[31] summarized common performance issues related to locks and proposed SyncPerf to detect lock misuse. In a related work, SyncProfiler[85] generates a graph to represent the wait relations between critical sections, and computes the performance impact of each critical section based on it. However, these approaches mainly focus on thread waiting due to lock contention, overlooking other events such as blocking IO, resources exhaustion, or other inter-thread dependencies. Addressing this, wPerf [91] records the OS's waiting events to cover all types of waiting in applications. It builds a graph to model the wait-for relationships among all threads, aiming to identify wait events that have significantly impact and are critical for total throughput. DepGraph[52] focuses on a single thread's performance, extracting the waiting dependency between the focused thread and all other system resources and threads. GAPP [71] adopts a different approach, measuring the cumulative concurrency rate of each thread to pinpoint those that may cause others to be blocked.

10.2 Blockchain Client Performance Measurement

Initial studies [34, 44, 49, 72, 82] mainly focused on measuring the overall performance (e.g., throughput, latency) of Blockchain clients. However, these broad metrics are unable to reflect the detailed performance of different modules, thus providing limited guidance for developers to locate performance bottlenecks. In response, Zheng et al.[88, 89] instrument the source code of clients to analyze time cost distributions, uncovering that the primary bottlenecks are located in transaction execution. Li et al.[64] investigated further into transaction execution, and find a significant consumption of time at the storage layer. From a different perspective, Perez et al.[73] and Aldweesh et al.[32] calculate the CPU time needed for each opcode during transaction execution. Inagaki et al.[61] applied the thread dependency graph to Hyperledger Fabric, identifying an issue with lock contention. Toyoda et al.[77] used pprof to measure the time taken by each function in client, aiming to identify the most time-consuming ones. Our tool is capable of identifying all the bottlenecks discovered by the previous work and also unearth some unknown bottlenecks.

10.3 Blockchain Client Optimization

To overcome the performance issue, numerous studies have focused on optimizing the blockchain clients. Some introduce new underlying consensus protocols with varied trade-offs to achieve high transaction throughput [36, 56]. Others try to enhance throughput by sharding transaction execution and the blockchain state [30, 46, 70, 79, 86]. Additionally, several works aim to devise solutions for the concurrent execution of smart contracts [33, 47, 84]. In the realm of DAG-based blockchain systems, transactions and blocks are organized at different vertices of the directed graph, enabling parallel block generation and inclusion [63, 65, 80]. Furthermore, both LVMT [64] and LMPTs [43] propose a novel MPT (Merkle Patricia Trie) design, which has the potential to reduce the costly read/write and disk I/O operations on the critical path of transaction processing.

11 Conclusion

This paper introduces a novel approach for detecting the manipulatable bottleneck in blockchain clients. We propose the two primary causes of bottlenecks in software and monitor these symptoms during clients runtime. To understand the client's execution and accurately characterize its behavior, we propose ThreadNeck, which models the client as a number of threads and delineates their interrelationships. Using ThreadNeck, we analyze four mainstream clients developed in different programming languages and identify 13 types of manipulatable bottlenecks. Our case studies show that ThreadNeck can identify problems other tool cannot find, and also maintain an low overhead.

Acknowledgment

We thank the anonymous reviewers for their helpful comments. This research is partially supported by the Hong Kong RGC Project (PolyU15222320), National Natural Science Foundation of China (No. 62172301).

References

[1] 2023. A java agent to generate method mappings to use with the linux perf tool. https://github.com/jvm-profiling-tools/perf-map-agent.
[2] 2023. Cryptocurrency Statistics 2023. https://www.forbes.com/advisor/au/investing/cryptocurrency/cryptocurrency\protect\discretionary{\char\hyphenchar\font}{}{}statistics/.
[3] 2023. dtrace. https://www.brendangregg.com/dtrace.html.
[4] 2023. An enterprise-grade Java-based, Apache 2.0 licensed Ethereum client. https://github.com/hyperledger/besu.
[5] 2023. ethernodes. https://www.ethernodes.org//.
[6] 2023. FISCO BCOS is a stable, efficient, and secure permissioned blockchain platform that has been widely used in real-world industry applications. https://github.com/FISCO-BCOS/FISCO-BCOS.
[7] 2023. gprof. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.
[8] 2023. Java thread memory calculation. https://stackoverflow.com/questions/67068623/java-thread-memory-calculation.
[9] 2023. jprofiler. https://www.ej-technologies.com/products/jprofiler/overview.html.
[10] 2023. Lifecycle and States of a Thread in Java. https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-java/.
[11] 2023. Modular, contributor-friendly and blazing-fast implementation of the Ethereum protocol, in Rust. https://github.com/paradigmxyz/reth.
[12] 2023. Most Commonly Used Blockchain Platforms of 2023. https://webisoft.com/articles/blockchain-platforms/.
[13] 2023. Multiple concurrent connections with Vertx. https://stackoverflow.com/questions/58361727/multiple-concurrent-connections-with-vertx.
[14] 2023. Official Go implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum.
[15] 2023. Perf. https://perf.wiki.kernel.org/index.php/Main_Page.
[16] 2023. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.
[17] 2023. Perf Sampling-Driven Performance Analysis Report. http://kernel.taobao.org/index.php/Documents/Perf_performance_analysis.
[18] 2023. pprof is a tool for visualization and analysis of profiling data. https://github.com/google/pprof.
[19] 2023. Run Queues and Wait Queues. https://cs4118.github.io/www/2023-1/lect/10-run-wait-queues.html.
[20] 2023. runtime/trace. https://pkg.go.dev/runtime/trace.
[21] 2023. Sampling CPU and HEAP profiler for Java. https://github.com/async-profiler/async-profiler.
[22] 2023. Scheduling In Go : Go Scheduler. https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html.
[23] 2023. The Ethereum network is currently undergoing a DoS attack. https://blog.ethereum.org/2016/09/22/ethereum-network-currently-undergoing-dos-attack.
[24] 2023. The Golang Scheduler. https://www.kelche.co/blog/go/golang-scheduling/.
[25] 2023. Thread Synchronization. https://www.techopedia.com/definition/24349/thread-synchronization.
[26] 2024. Ethereum Clients' Node Syncing Methods. https://etherworld.co/2022/07/21/ethereum-clients-node-syncing-methods/.
[27] 2024. Guide to decentralized app. https://aglowiditsolutions.com/blog/decentralized-application-dapp-guide/.
[28] 2024. Hyperledger Besu for private networks. https://besu.hyperledger.org/private-networks.
[29] 2024. What's a blockchain node? https://worldcoin.org/articles/what-is-a-blockchain-node.
[30] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).
[31] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*. 298–313.
[32] Amjad Aldweesh, Maher Alharby, Maryam Mehrnezhad, and Aad van Moorsel. 2021. The OpBench Ethereum opcode benchmark framework: Design, implementation, validation and experiments. *Performance Evaluation* 146 (2021), 102168.
[33] Parwat Singh Anjana, Sweta Kumari, Sathya Peri, Sachin Rathor, and Archit Somani. 2019. An efficient framework for optimistic concurrent execution of smart contracts. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 83–92.
[34] Arati Baliga, Nitesh Solanki, Shubham Verekar, Amol Pednekar, Pandurang Kamat, and Siddhartha Chatterjee. 2018. Performance characterization of hyperledger fabric. In *2018 Crypto Valley conference on blockchain technology (CVCBT)*. IEEE, 65–74.
[35] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014), 2–1.
[36] Vitalik Buterin, Daniël Reijsbergen, Stefanos Leonardos, and Georgios Piliouras. 2020. Incentives in Ethereum's hybrid Casper protocol. *International Journal of Network Management* 30, 5 (2020), e2098.
[37] Si Chen, Jinyu Zhang, Rui Shi, Jiaqi Yan, and Qing Ke. 2018. A comparative testing on performance of blockchain and relational database: Foundation for applying smart technology into current business systems. In *Distributed, Ambient and Pervasive Interactions: Understanding Humans: 6th International Conference, DAPI 2018, Held as Part of HCI International 2018, Las Vegas, NV, USA, July 15–20, 2018, Proceedings, Part I 6*. Springer, 21–34.
[38] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. 2017. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *Information Security Practice and Experience: 13th International Conference, ISPEC 2017, Melbourne, VIC, Australia, December 13–15, 2017, Proceedings 13*. Springer, 3–24.
[39] Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. 2020. Understanding ethereum via graph analysis. *ACM Transactions on Internet Technology (TOIT)* 20, 2 (2020), 1–32.
[40] Weimin Chen, Xiapu Luo, Haipeng Cai, and Haoyu Wang. 2024. Towards Smart Contract Fuzzing on GPU. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 195–195.
[41] Wubing Chen, Zhiying Xu, Shuyu Shi, Yang Zhao, and Jun Zhao. 2018. A survey of blockchain applications in different domains. In *Proceedings of the 2018 International Conference on Blockchain Technology and Application*. 17–21.
[42] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. 2023. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2517–2532.
[43] Jemin Andrew Choi, Sidi Mohamed Beillahi, Peilun Li, Andreas Veneris, and Fan Long. 2022. LMPTs: Eliminating Storage Bottlenecks for Processing Blockchain Transactions. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–9.
[44] Wonseok Choi and James Won-Ki Hong. 2021. Performance evaluation of ethereum private and testnet networks using hyperledger caliper. In *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE, 325–329.
[45] Charlie Curtsinger and Emery D Berger. 2015. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 184–197.
[46] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*. 123–140.
[47] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Adding concurrency to smart contracts. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 303–312.
[48] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. 2018. Untangling blockchain: A data processing view of blockchain systems. *IEEE transactions on knowledge and data engineering* 30, 7 (2018), 1366–1385.
[49] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM international conference on management of data*. 1085–1100.
[50] Zhongli Dong, Emma Zheng, Young Choon, and Albert Y Zomaya. 2019. Dagbench: A performance evaluation framework for dag distributed ledgers. In *2019 IEEE 12th international conference on cloud computing (CLOUD)*. IEEE, 264–271.
[51] Francois Doray and Michel Dagenais. 2016. Diagnosing performance variations by comparing multi-level execution traces. *IEEE Transactions on Parallel and Distributed Systems* 28, 2 (2016), 462–474.
[52] Naser Ezzati-Jivan, Quentin Fournier, Michel R Dagenais, and Abdelwahab Hamou-Lhadj. 2020. Depgraph: Localizing performance bottlenecks in multi-core applications using waiting dependency graphs and software tracing. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 149–159.
[53] Gang Fan, Rongxin Wu, Qingkai Shi, Xiao Xiao, Jinguo Zhou, and Charles Zhang. 2019. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 72–82.
[54] Jair Ferreira, Eduardo Carvalho, Bruno V Ferreira, Cleidson de Souza, Yoshihiko Suhara, Alex Pentland, and Gustavo Pessin. 2017. Driver behavior profiling: An investigation with different smartphone sensors and machine learning. *PLoS one* 12, 4 (2017), e0174959.
[55] Fabien Geyer, Holger Kinkelin, Hendrik Leppelsack, Stefan Liebald, Dominik Scholz, Georg Carle, and Dominic Schupke. 2019. Performance perspective on private distributed ledger technologies for industrial networks. In *2019 International Conference on Networked Systems (NetSys)*. IEEE, 1–8.
[56] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*. 51–68.
[57] Richard Greene and Michael N Johnstone. 2018. An investigation into a denial of service attack on an ethereum network. (2018).

[58] Brendan Gregg. 2014. *Systems performance: enterprise and the cloud.* Pearson Education.
[59] Zheyuan He, Zihao Li, Ao Qiao, Xiapu Luo, Xiaosong Zhang, Ting Chen, Shuwei Song, Dijun Liu, and Weina Niu. 2024. NURGLE: Exacerbating Resource Consumption in Blockchain State Storage via MPT Manipulation. *arXiv preprint arXiv:2406.10687* (2024).
[60] Hwanjo Heo, Seungwon Woo, Taeung Yoon, Min Suk Kang, and Seungwon Shin. 2023. Partitioning Ethereum without Eclipsing It.. In *NDSS*.
[61] Tatsushi Inagaki, Yohei Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2019. Profile-based detection of layered bottlenecks. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 197–208.
[62] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P Wright, and Erez Zadok. 2006. Operating System Profiling via Latency Analysis.. In *OSDI*, Vol. 6. 89–102.
[63] Colin LeMahieu. 2018. Nano: A feeless distributed cryptocurrency network. *Nano [Online resource]. URL: https://nano. org/en/whitepaper (date of access: 24.03. 2018)* 16 (2018), 17.
[64] Chenxing Li, Sidi Mohamed Beillahi, Guang Yang, Ming Wu, Wei Xu, and Fan Long. 2023. LVMT: An Efficient Authenticated Storage for Blockchain. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 135–153.
[65] Chenxing Li, Peilun Li, Dong Zhou, Wei Xu, Fan Long, and Andrew Yao. 1805. Scaling nakamoto consensus to thousands of transactions per second, 2018. *arXiv preprint arXiv:1805.03870* (1805).
[66] Kai Li, Jiaqi Chen, Xianghong Liu, Yuzhe Richard Tang, XiaoFeng Wang, and Xiapu Luo. 2021. As Strong As Its Weakest Link: How to Break Blockchain DApps at RPC Service.. In *NDSS*.
[67] Kai Li, Yibo Wang, and Yuzhe Tang. 2021. Deter: Denial of ethereum txpool services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1645–1667.
[68] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaoze Ni, Wenwu Yang, Xi Chen, and Ting Chen. 2023. Demystifying defi mev activities in flashbots bundle. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 165–179.
[69] Yi Liu, Yunchun Li, Honggang Zhou, Jingyi Zhang, Hailong Yang, and Wei Li. 2018. A fine-grained performance bottleneck analysis method for HDFS. In *Network and Parallel Computing: 15th IFIP WG 10.3 International Conference, NPC 2018, Muroran, Japan, November 29–December 1, 2018, Proceedings 15*. Springer, 159–163.
[70] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. 2016. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 17–30.
[71] Reena Nair and Tony Field. 2020. Gapp: A fast profiler for detecting serialization bottlenecks in parallel linux applications. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 257–264.
[72] Qassim Nasir, Ilham A Qasse, Manar Abu Talib, Ali Bou Nassif, et al. 2018. Performance analysis of hyperledger fabric platforms. *Security and Communication Networks* 2018 (2018).
[73] Daniel Perez and Benjamin Livshits. 2019. Broken metre: Attacking resource metering in EVM. *arXiv preprint arXiv:1909.07220* (2019).
[74] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and David Mohaisen. 2020. Exploring the attack surface of blockchain: A comprehensive survey. *IEEE Communications Surveys & Tutorials* 22, 3 (2020), 1977–2008.
[75] Massimiliano Taverna and Kenneth G Paterson. 2023. Snapping snap sync: practical attacks on go Ethereum synchronising nodes. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3331–3348.
[76] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. 2018. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th international symposium on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS)*. IEEE, 264–276.
[77] Kentaroh Toyoda, Koji Machi, Yutaka Ohtake, and Allan N Zhang. 2020. Function-level bottleneck analysis of private proof-of-authority ethereum blockchain. *IEEE Access* 8 (2020), 141611–141621.
[78] Robert Van Mölken. 2018. *Blockchain across Oracle: Understand the details and implications of the Blockchain for Oracle developers and customers.* Packt Publishing Ltd.
[79] Gang Wang, Zhijie Jerry Shi, Mark Nixon, and Song Han. 2019. Sok: Sharding on blockchain. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 41–61.
[80] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. 2020. SoK: Diving into DAG-based blockchain systems. *arXiv preprint arXiv:2012.06128* (2020).
[81] Yibo Wang, Wanning Ding, Kai Li, and Yuzhe Tang. 2023. Understanding Ethereum Mempool Security under Asymmetric DoS by Symbolic Fuzzing. *arXiv preprint arXiv:2312.02642* (2023).
[82] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. 2017. On availability for blockchain-based systems. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 64–73.
[83] Shuohan Wu, Zihao Li, Luyi Yan, Weimin Chen, Muhui Jiang, Chenxu Wang, Xiapu Luo, and Hao Zhou. 2024. Are we there yet? unraveling the state-of-the-art smart contract fuzzers. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.
[84] Lian Yu, Wei-Tek Tsai, Guannan Li, Yafe Yao, Chenjian Hu, and Enyan Deng. 2017. Smart-contract execution with concurrent block building. In *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 160–167.
[85] Tingting Yu and Michael Pradel. 2016. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 389–400.
[86] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 931–948.
[87] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. Deepinfer: Deep type inference from smart contract bytecode. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 745–757.
[88] Peilin Zheng, Quanqing Xu, Xiapu Luo, Zibin Zheng, Weilin Zheng, Xu Chen, Zhiyuan Zhou, Ying Yan, and Hui Zhang. 2022. Aeolus: Distributed execution of permissioned blockchain transactions via state sharding. *IEEE Transactions on Industrial Informatics* 18, 12 (2022), 9227–9238.
[89] Peilin Zheng, Quanqing Xu, Xiapu Luo, Zibin Zheng, Weilin Zheng, Xu Chen, Zhiyuan Zhou, Ying Yan, and Hui Zhang. 2023. State Sharding for Permissioned Blockchain. In *Blockchain Scalability*. Springer, 143–164.
[90] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain challenges and opportunities: A survey. *International journal of web and grid services* 14, 4 (2018), 352–375.
[91] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. 2018. wPerf: Generic Off CPU Analysis to Identify Bottleneck Waiting Events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 527–543.

Received 2024-04-12; accepted 2024-07-03