Received July 9, 2020, accepted July 21, 2020, date of publication July 24, 2020, date of current version August 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3011876

Function-Level Bottleneck Analysis of Private Proof-of-Authority Ethereum Blockchain

KENTAROH TOYODA[1,2], (Member, IEEE), KOJI MACHI[3,4], YUTAKA OHTAKE[4], AND ALLAN N. ZHANG[1], (Associate Member, IEEE)

[1]Agency for Science, Technology, and Research (A*STAR), Singapore Institute of Manufacturing Technology (SIMTech), Singapore 138634
[2]Faculty of Science and Technology, Keio University, Yokohama 223-8522, Japan
[3]Londobell Labs Pte. Ltd., Singapore 049319
[4]SingulaNet Ltd., Minato City 106-0046, Japan

Corresponding author: Kentaroh Toyoda (kentaroh.toyoda@ieee.org)

This work was supported in part by the Performance Analysis of Permissioned Ethereum Blockchain Industry Project under Grant SC26/19-331190, and in part by the MEXT/JSPS, Japan, under Grant KAKENHI 18K18162.

ABSTRACT Private Ethereum blockchain-based systems are demanded in many industry sectors. However, the throughput performance of these systems does not meet their expectations. Many researchers have analyzed the performance of private blockchains, but their studies have failed to analyze root causes. In this paper, we perform a deep function-level bottleneck analysis for the private Ethereum blockchain. As the Ethereum client application is developed with golang, we leverage `pprof`, which is a resource-profiling tool for golang, and custom golang functions to measure the time taken by functions. To easily configure parameters and conduct our test, we code a shell script that automates the building process of a private Ethereum blockchain with docker containers. We conducted a series of experiments and identified the bottleneck function that is called every time a transaction arrives at an Ethereum node. In addition, we also found that the multi-threading is not well utilized, meaning that there is much room for improvement.

INDEX TERMS Ethereum, private blockchain, performance evaluation.

I. INTRODUCTION

An increasing number of blockchain-based systems have emerged in recent years; according to one report, more than 90 Ethereum-based systems have been launched so far [1]. For example, Augur is an Ethereum-based decentralized prediction market that leverages blockchain's functionalities such as smart contracts and cryptocurrency.[1] However, such a system is suited to cases in which data and information are "open." Recently, private blockchains, for example, private Ethereum and Hyperledger Fabric, in which only authoritative parties manage blockchains, have been chosen for multi-site systems that shared among multiple stakeholders. Such systems bring time-stamping, tamper-proof and proof-of-existence aspects to enterprise systems.

Although the transaction verification speed of private blockchains is typically much faster than that of public ones, it remains far below the level that enterprises demand. Many researchers have analyzed private blockchains in terms of several performance metrics such as latency, throughput and resource utilization, for example, [2]–[4]. Dinh *et al.* analyzed several performance metrics with Blockbench, a suite of analysis tools for private Ethereum, Parity and Hyperledger blockchains [2], [3]. However, the root of bottlenecks has not been analyzed. In order to improve the performance, it is necessary to determine what functions and procedures are at the root of bottlenecks and how much time and resources are spent on the bottlenecks.

In this paper, we aim to determine the function-level bottleneck of the private Ethereum blockchain. For this, we leverage `pprof`, which is a resource-profiling tool for golang, since the Ethereum client, `geth`, is developed with golang. With `pprof`, we can gain useful performance metrics, such as processed time, number of created threads, consumed memory size and blocking by functions. By analyzing the result of `pprof`, we narrow down the bottleneck functions in `geth` and derive more detailed metrics that `pprof` cannot output with our logging functions. In addition, we code a shell script that easily builds and tests a private Ethereum blockchain with different parameter settings.

The associate editor coordinating the review of this manuscript and approving it for publication was Noor Zaman.

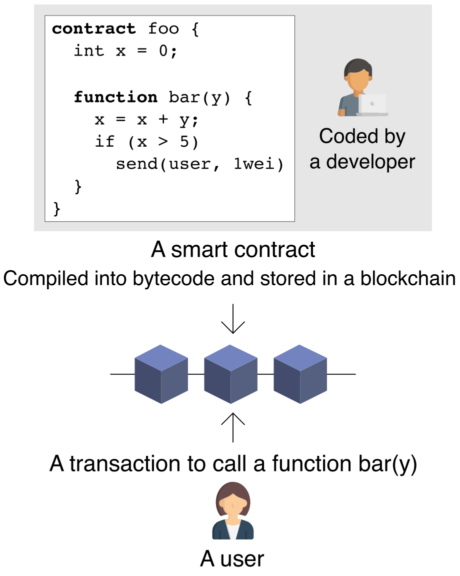
[1]https://www.augur.net/

We conduct a series of experiments with workstations and test both normal transactions and transactions to execute functions in several basic smart contracts. As a result, we identify a bottleneck function that is called every time a transaction arrives at an Ethereum node. Based on the results, it is necessary to find ways to improve the performance, such as pipelining and replacing the bottleneck functions. For this, we also show that the multi-threading functions are implemented in `geth`, meaning that there is much room for improvement.

The contribution of this paper is three-fold.

1) We have developed a series of codes to easily configure parameters and build and test a private Ethereum blockchain.
2) We have disclosed the developed toolset at our github repository so that other researchers can reproduce our results.
3) We have identified the bottleneck functions in a private Ethereum blockchain by analyzing the results of our experiments.

The rest of the paper is organized as follows: The fundamentals of Ethereum are presented in Section II. Related work is summarized in Section III. The proposed method is described in Section IV. Performance analysis is conducted in Section V. The conclusions and future work are presented in Section VII.



```
contract foo {
  int x = 0;

  function bar(y) {
    x = x + y;
    if (x > 5)
      send(user, 1wei)
  }
}
```

Coded by
a developer
A smart contract
Compiled into bytecode and stored in a blockchain
A transaction to call a function bar(y)
A user

FIGURE 1. An illustration of a smart contract and a transaction to execute a function within it.

II. ETHEREUM BLOCKCHAIN

Ethereum is a blockchain-enabled decentralized open-source platform for decentralized systems [5]. On Ethereum, developers can write codes (smart contracts) that control digital currency (ether) and run smart contracts via transaction execution. FIGURE 1 illustrates a toy example of a smart contract and of code execution via a transaction. In this example, a developer designed a smart contract code *foo*, which stores integer x, in which users can update its value as $x = x + y$ and obtain 1 `wei` (the smallest denomination of ether) if x is greater than five by calling the function of $bar(y)$ via transaction. Any programmable procedures, that is, calculation, data storing and cryptocurrency transfer, can be realized with a smart contract. In Ethereum, the JavaScript-like programming language Solidity is often used to code smart contracts.

In the case of many businesses, there is a demand to leverage the features of blockchain, for example, time-stamping and tamper-proofing, for which purposes the private blockchain is often demanded, since transactions should be kept private in most business cases. Ethereum is developed for both public and private blockchains. The public Ethereum is operated by a large number of voluntary nodes, and block creation is done by miners. Hence, the consensus among them is achieved with Proof-of-Work (PoW), which is similar to Bitcoin. Proof-of-Stake (PoS) is another consensus algorithm designed for public Ethereum that allows miners with more cryptocurrency to mine blocks more easily than others with less cryptocurrency.

Private Ethereum, meanwhile, is designed to allow limited users to access a shared blockchain. Hence, some authoritative entities manage a blockchain, which is grounded in a more relaxed assumption than the public one. Hence, the consensus is achieved by PoA [6] in the form of the Clique algorithm, which requires much less computation than PoW by eliminating the "crypto-puzzle" in PoW. In PoA, the task of block creation is executed by predetermined authorized nodes, that is, *sealers*.

Ethereum is an open-source software written in golang[2] whose code is easy to analyze and modify. Once the codes are compiled, a command-line interface called `geth` is generated. `geth` can be run on major operation systems such as Linux, macOS and Windows.

III. RELATED WORK

Blockchain-enabled systems are used in many fields, including finance, logistics and the biomedical industry [10]. This means that such systems may require rapid transaction processes and rigorous security and privacy measures. Many researchers have thus studied the performance of blockchains in terms of security aspects. In the following, we summarize this research in terms of two discrete topics: performance measurement and smart contract analysis.

A. PERFORMANCE MEASUREMENT

Dinh *et al.* developed Blockbench, a suite of analysis tools for private Ethereum, Parity and Hyperledger blockchains [2], [3]. They evaluated several performance metrics, including throughput and latency, with the tools that they developed. Furthermore, nine smart contracts, for example, a key-value

[2]The source code can be obtained from https://github.com/ethereum

TABLE 1. Comparison of state-of-the-art studies that include private Ethereum and our study.

Ref.	Blockchains	Performance Metrics			Types of Transactions		Function-level Analysis
		Latency	Throughput	Resource	Normal	Smart Contract	
[2], [3]	Ethereum (PoW), Parity (PoA) and Hyperledger	✓	✓	✓		✓	
[4]	Ethereum (PoA) and Parity	✓			✓		
[7]	Ethereum (PoW) and Hyperledger Fabric	✓	✓			✓	
[8]	Ethereum (PoW)		✓		✓		
[9]	Ethereum (PoW and PoA)	✓	✓		✓		
Our study	Ethereum (PoA)	✓	✓	✓	✓	✓	✓

store and name-registrar contract, were implemented and compared.

Loghin *et al.* analyzed the energy-efficient aspect of blockchains [11]. They evaluated throughput-to-consumed energy with three low-power computers, namely Raspberry Pi 3, NVIDIA's Jetson TX2, and a high performance computer with an Intel Xeon processor. Their analysis revealed several interesting facts. For example, blockchain nodes operating on Jetson TX2 achieved around 80% and 30% of the throughput of Parity and Hyperledger, respectively, while using 18$\times$ and 23$\times$ less energy, respectively, compared to powerful servers with the Intel Xeon processor.

Bez *et al.* analyzed the throughput of PoW-based Ethereum [8]. They set up a local Ethereum blockchain and measured the throughput by varying the number of miners from 1 to 8. They showed that the average number of transactions was about 8.17 and 11.07 transactions per second (TPS) for 1 and 8 miners, respectively.

Hao *et al.* analyzed the latency and throughput of private Ethereum and Hyperledger Fabric blockchains [7]. They chose PoW and PBFT as consensus algorithms of Ethereum and Hyperledger Fabric, respectively. They conducted an experiment to determine how the throughput was improved by skipping the consensus process, and their results showed that the throughput was almost doubled.

Rouhani and Deters measured transaction process time for private Ethereum and Parity [4]. Similarly, Pongnumkul *et al.* measured the latency and throughput of smart-contract execution in private Ethereum and Hyperledger Fabric [12]. They tested three basic smart contracts: (i) account creation, (ii) mint and (iii) coin transfer for both blockchains.

Schäffer *et al.* clarified the effects of parameter choices for private Ethereum [9]. More specifically, throughput, latency and scalability are evaluated when varying the parameters such as (i) block interval, (ii) block size, (iii) number of nodes and (iv) computational resources (CPU, memory and network bandwidth). The performance was tested in PoW and PoA consensus algorithms and two different smart contracts. They concluded that the bottlenecks of a network are non-trivial factors in the blockchain.

Kim *et al.* developed NodeFinder, a measurement tool for the Ethereum network [13]. They measured the performance of a network side that affects information propagation delay and a consensus algorithm.

The above papers pertain to Ethereum and other blockchain networks; the following papers only deal with Hyperledger Fabric.

Baliga *et al.* evaluated the performance and scalability features of Hyperledger Fabric version 1.0 [14]. They performed a series of experiments, to measure the throughput and latency characteristics of the blockchain by subjecting it to the different sets of workloads.

Nasir *et al.* conducted a performance analysis of the two versions of Hyperledger Fabric, v0.6 and v1.0 [15]. The performance evaluation was assessed in terms of execution time, latency and throughput, by varying the workload in each platform up to 10,000 transactions. The authors also analyzed the scalability of the two platforms by varying the number of nodes up to 20 nodes. They concluded that although Hyperledger Fabric v1.0 consistently outperformed Hyperledger Fabric v0.6, its performance did not reach the performance level of traditional database systems under high-workload scenarios.

Sukhwani *et al.* presented a performance model of Hyperledger Fabric v1.0+ using Stochastic Reward Nets (SRN) [16]. The authors used Hyperledger Caliper as the evaluation platform and clarified that the performance bottleneck of the ordering service, which is nearly equivalent to sealing in private Ethereum, and writing data into a blockchain could be mitigated using a larger block size by sacrificing latency.

Thakkar *et al.* analyzed the impact of various configuration parameters, for example, block size and resource allocation, on throughput and latency in Hyperledger Fabric v1.0 [17]. The authors provided six guidelines on configuring parameters to attain the maximum performance. They introduced three optimizations to improve the overall performance by 16× (i.e., from 140 TPS to 2,250 TPS).

Ampel *et al.* presented a performance analysis of the Hyperledger Sawtooth blockchain [18]. The authors showed that throughput can be improved by increasing the number of transactions in a block up to a certain point. They also analyzed the relationships among transaction rates, latency and memory consumption.

B. ANALYSIS OF SMART CONTRACT

The above papers mainly deal with performance analysis regarding throughput and latency in different scenarios. Many papers have analyzed the contents of smart contracts and their transactions.

Furthermore, Bartoletti *et al.* revealed that Ethereum's smart contacts were often misused for Ponzi schemes, also known as high-yielding investment programs [19]. They traced the flow of Ponzi-scheme-related money collected by their open-source tool and analyzed the smart contracts of collected Ponzi schemes. Chen *et al.* also proposed a method of detecting Ponzi schemes in Ethereum using machine-learning technique [20]. They used user accounts information and smart-contract codes to calculate features which are used to train supervised machine-learning classifiers.

Torres *et al.* investigated another type of smart contract that lures Ethereum users, namely, *honeypots* [21]. Honeypots are smart contracts that involve an obvious flaw that allows a user to steal Ethereum from the contract by transferring some amount of Ethereum to the contract. However, a trapdoor is actually set, and once the user tries to exploit this vulnerability, it prevents the attempt from succeeding. The researchers developed a tool called HoneyBadger to automatically detect honeypot smart contracts in Ethereum, and they identified 690 such smart contracts, which had collected more than $90,000 for the honeypot creators.

The above examples reveal that Ethereum's smart-contract analyzers are necessary to detect and avoid attacks against smart contracts. Many useful tools of analysis are available, for example, [22]–[24].

C. LIMITATION OF THE EXISTING STUDIES

TABLE 1 lists the summary and comparison of existing research on private blockchains and our study. Although many researchers clarified performance metrics in various scenarios as listed in this table, no *root cause analysis* has been done. In other words, important facts, for example, what functions are the bottlenecks of performance and how much they affect performance, have not been investigated. Hence, the objective of this paper is to clarify function-level bottlenecks in private Ethereum. We chose private Ethereum as our target since it is one of the most successful private blockchains.

IV. PROPOSED METHOD

To perform the function-level bottleneck analysis, we first leverage `pprof`, which is a golang tool to measure CPU,

```go
// go-ethereum/log/benchmark.go
package log

import (
  "time"
  "runtime"
)

// GetFuncName: Return the function name
func GetFuncName() string {
    pc, _, _, _ := runtime.Caller(1)
    return runtime.FuncForPC(pc).Name()
}

// ElapsedTimeInMicroSec: Output a function'
    s process time to a geth log
func ElapsedTimeInMicroSec(t time.Time, name
     string) {
    elapsed := int64(time.Since(t) / time.
        Microsecond)
    Info("TIME:", "func", name, "elapsed",
        elapsed)
}
```

```go
// Insert the following codes into any
    function to be measured
import (
    "time"
    ...

    "github.com/ethereum/go-ethereum/log"
    ...
)

func Foo() ([]byte, error) {
    defer log.ElapsedTimeInMicroSec(
      time.Now(), log.GetFuncName())
    ...
}
```

FIGURE 2. Golang code snippets to measure process time to a `geth` log.

memory and input/output (I/O) utilization, blocking time and mutex by functions. The root causes of bottleneck and function calls can be found by analyzing the results of `pprof`. However, `pprof` itself cannot provide more detailed statistics, such as the time taken to process a transaction. Hence, we inserted the codes shown in FIGURE 2 to measure the process time of `geth`'s functions in the least intrusive manner possible. This specially crafted `geth` was installed into sealer nodes.

A. SYSTEM OVERVIEW

FIGURE 3 provides an overview of our testbed design. Our testbed is composed of three entities: i) bootnode, ii) sealer and iii) sender. Through a bootnode, nodes can join the specified private Ethereum blockchain and find each other. A sealer is an authoritative node that verifies transactions, seals blocks and broadcasts them to other nodes, while a sender is a node that sends transactions into the blockchain. A bootnode and N_{seal} ($N_{\mathrm{seal}} \geq 3$) sealers are deployed to a workstation, and the sealers manage a blockchain. Similarly,

TABLE 2. **Example of `geth` log.**

NODE	DATETIME	EVENT	DETAIL
...	...	...	...
sealer-2	[2020-02-12 04:53:46.440]	Broadcast transaction	hash=c97f16dfb00b recipients=2
sealer-3	[2020-02-12 04:53:46.440]	TIME:	func=github.com/ethereum/go-ethereum/p2p.(*Peer).handle elapsed=133
sealer-2	[2020-02-12 04:53:46.440]	Pooled new future transaction	hash=...from=0x...to=0x...
sealer-1	[2020-02-12 04:53:46.440]	TIME:	func=github.com/ethereum/go-ethereum/p2p.(*Peer).handle elapsed=130
...	...	...	...

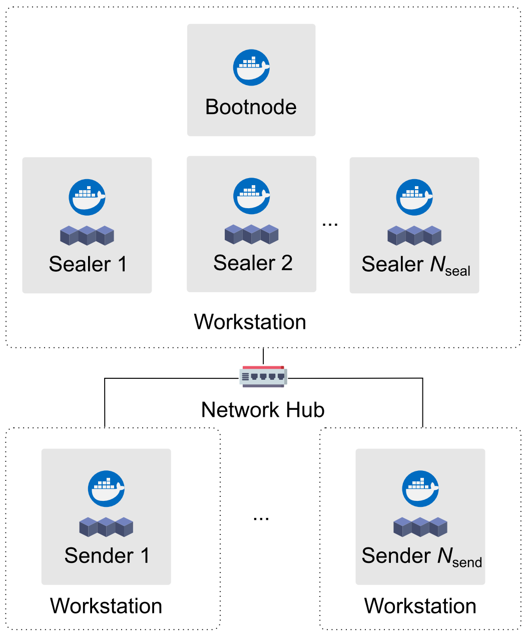


FIGURE 3. **System configuration for performance analysis.**

N_{send} ($N_{send} \geq 1$) senders are deployed to other workstations. The `web3` package of JavaScript (JS) is used to allow senders to send transactions. Sealers and senders execute `geth` to run a private Ethereum client.

`Docker` and `docker-compose` are used to easily deploy a series of nodes to workstations.[3] Since network latency has been well investigated in some papers, for example, [9] and [13], we try to eliminate its effect. Hence, three workstations are connected through 1-Gigabit Ethernet in a local network. Clock drift among workstations might cause inaccurate measurement. In order to avoid this, the system time of all the workstations was synchronized with a common network time protocol server pool.ntp.org. The following operations are executed in order with a shell script:

1) Set parameters, for example `geth` and the number of nodes, the type of transactions to be sent.
2) Build docker containers of a bootnode, sealers and senders.
3) Deploy them and start `geth` in each container.
4) Wait until every `geth` client has been started and synced.
5) Start `pprof` and `docker stats` logging.
6) Send N_{tx} transactions from senders.
7) Forward all log files in a workstation.
8) Stop every container.

[3]A series of codes are disclosed at https://github.com/kentaroh-toyoda/private_ethereum_performance_analysis

B. TYPES OF LOG

After deployment, four types of logging are started.

1) `geth` log: TABLE 2 shows an example of this. The output of logs in the form of functions in `geth` are chronologically stored by nodes. We can extract useful information such as the time taken by functions, when transactions and blocks are propagated and when senders have sent each transaction.
2) JS log: The output of JS log by sender. As shown in TABLE 3, a datetime is recorded for each successfully sent transaction with its nonce and hash values. By combining this and the `geth` log, we can calculate the latency using each sent transaction and how each transaction is sealed in blocks.
3) `pprof` log: `pprof` is a versatile profiling tool, and one can specify the type of resource to be monitored. For example, if the CPU profile is specified, `pprof` monitors the utilization of CPU for 30 seconds and records the cumulative process time of executed functions. A log file is stored as the pb.gz file format and can be output and visualized in several ways, in text and in graph, as shown later in FIGURES 11 and 12.
4) `docker stats` log: Sealers' resource utilization, for example, CPU, memory and network I/O, are traced.

TABLE 5 shows the list of logs and nodes. As a bootnode is only responsible for managing node information, only a `geth` log is collected, and it is used simply to verify that the blockchain network works. Sealers store the log files of `geth`, `pprof` and `docker stats` so that function-level root cause analysis is performed. By contrast, as we are interested in the latency of sending transactions, senders store `geth`, JS and `docker stats` logs.

C. TYPES OF TRANSACTION AND SMART CONTRACT

There are two types of transactions: i) a normal transaction that sends an `ether` from an account to another and ii) a

TABLE 3. Example of sent transactions log.

DATETIME	NONCE	DETAIL	TX HASH
2020-02-12 13:56:39.759	0	success	0xc97f16221b500d3352c2467fbc2ed43cda95b934fa604a4cfbe80723f5dfb00b
2020-02-12 13:56:39.762	1	success	0x02b0a87f4fa2b10f931d7b2732613115da0c6d2e9d2b54479b7bc6f07adb17af
...	...	...	...

TABLE 4. Example of `docker stats` log.

DATETIME	CONTAINER ID	NAME	CPU	MEM USAGE / LIMIT	MEM	NET I/O	BLOCK I/O
...	...	...	...	...	...	...	...
2020-02-12 04:53:45	6df79f157bf4	sealer-2	13.98%	1.883GiB / 8GiB	23.54%	92.5kB / 92.9kB	0B / 188kB
2020-02-12 04:53:45	bd04db6c081c	sealer-3	13.43%	370.7MiB / 8GiB	4.53%	110kB / 111kB	0B / 188kB
2020-02-12 04:53:45	a2eb6a130207	sealer-1	15.16%	1.894GiB / 8GiB	23.68%	93.5kB / 87.8kB	0B / 188kB
...	...	...	...	...	...	...	...

TABLE 5. List of logs and nodes.

	BOOTNODE	SEALER	SENDER
`geth` log	✓	✓	✓
JS log			✓
`pprof` log		✓	
`docker stats` log		✓	✓

transaction to execute a smart contract. The following four types of smart contracts are prepared[4]:

1) `DoNothing`: A program that does not execute anything.
2) `CPUheavy`: A program of quick sort with a given number.
3) `KVstore`: A program of a key-value data storage.
4) `FungibleToken`: A simple token management application that consists of two functions: (i) `mint(value)`, in which tokens are created by `value`, and (ii) `burn(value)`, in which tokens are subtracted by `value`.

V. PERFORMANCE EVALUATION

A series of measurements were conducted with the shell script described in the previous section and the parameters listed in TABLE 6. Resources (# of CPU and RAM) assigned to each sealer container were varied to determine the effect on the performance. We first analyzed performance metrics, and then bottlenecks. Miner.gaslimit and miner.gastarget are the maximum and target values of total gas consumption in a block, which means that we can roughly control the number of transactions involved in a block. Since we do not want to limit the number of transactions in a block, these values are set sufficiently high. For the same reason, the number of pooled transactions to be sealed can be controlled by the

[4]`DoNothing`, `KVstore` and `CPUheavy` are distributed in https://github.com/ooibc88/blockbench/tree/master/benchmark/contracts/ethereum while `FungibleToken` is found in https://github.com/kentaroh-toyoda/private_ethereum_performance_analysis/tree/master/sender-sc/tools/smart_contracts/FungibleToken.

TABLE 6. Parameters for performance evaluation. All workstations have the same specification in terms of OS, CPU and RAM.

PARAMETER	VALUE
OS	Ubuntu 18.04.04
CPU	32-core Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
Total RAM	128 GB
# of CPU assigned to sealers	1, 2, 4 or 8 each (Default: 4)
RAM assigned to sealers	1, 2, 4 or 8GB each (Default: 4GB)
# of sealers (N_{seal})	3, 4, 6 or 8 (Default: 3)
Block interval	1, 2, 4, 8 sec (Default: 1 sec)
# of senders (N_{send})	2
# of transactions	5,000 each (10,000 from the two senders)
# of concurrencies in `core/tx_cacher.go`	1, 2, 4, 8, 16 and 32 (Default: 8)
Geth version	1.19.10-stable
miner.gaslimit	4,294,967,296
miner.gastarget	4,294,967,296
txpool.globalslots	65,536
txpool.accountslots	65,536
txpool.accountqueue	65,536
txpool.globalqueue	65,536
# of trials	Repeated until at least 100 results were obtained

series of parameters of txpool.*, and they are all set to 65,536, which means that our setting is far enough to accommodate tens of thousands of transactions in a block. The evaluation was repeated so that at least 100 results were obtained.

A. PERFORMANCE ANALYSIS

The performance metrics that we measured were i) throughput, ii) latency and iii) resource utilization. Throughput is defined as the number of successfully processed transactions per second. Latency, meanwhile, is defined as the time elapsed for a specific process according to the evaluated item. We measured CPU, memory and network I/O with `docker stats`. A boxplot is primarily used to visualize statistical data, as it can easily display the distribution of data [25].

Before analyzing function bottlenecks, we measured the latency of transaction submission. In this experiment, we used

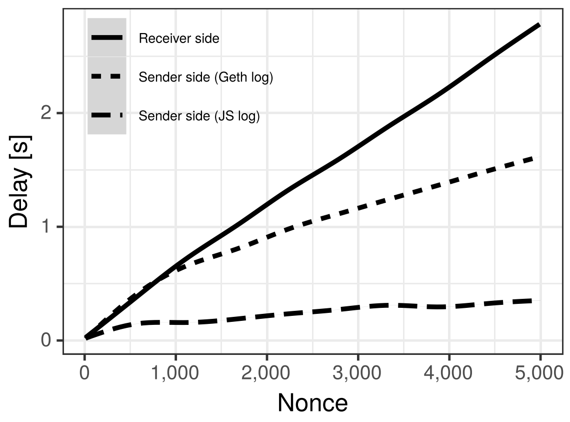


FIGURE 4. Latency of transactions (normal transactions, $N_{\text{CPU}} = 4$, $M = 4$ GB, $B = 1$ s).

only one sealer, and a total of 5,000 transactions were sent out. Hence, a nonce, which is a sequential index used to generate a transaction, was incremented from 0 to 4,999. Transactions were generated through a `web3` package in JS and are sent out via senders' `geth`. The transactions were then verified by sealers and propagated among them. Hence, the first delay occurred when transactions were generated. This could be measured from senders' `geth` logs. The second one occurred when the transactions were sent out by senders' `geth` client, which could be measured through JS log files. The last one could be measured at the sealers' side and was defined as the time difference between when a transaction was received by a sealer for the first time and when it was sent out. FIGURE 4 shows the delay associated with these three types. This figure indicates that it took about 0.5 seconds to generate 5,000 transactions and 1.7 seconds for a sender to send out them. In addition, it took approximately 2.8 seconds for a sealer to receive 5,000 transactions. As sealers must process not only transactions but also other tasks, such as block generation, more delay was accumulated over time.

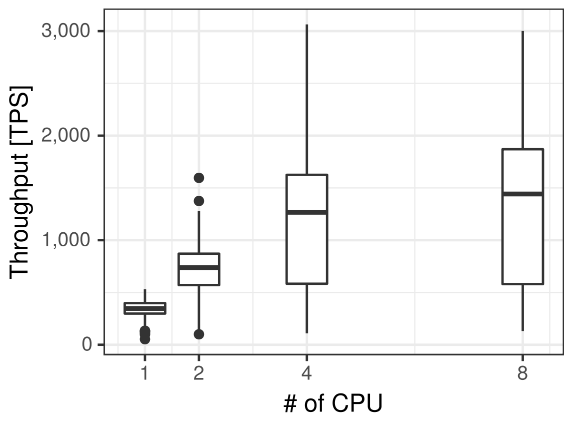


FIGURE 5. Throughput versus the number of CPU assigned to each sealer (normal transactions, $N_{\text{seal}} = 3$, $M = 4$ GB, $B = 1$ s).

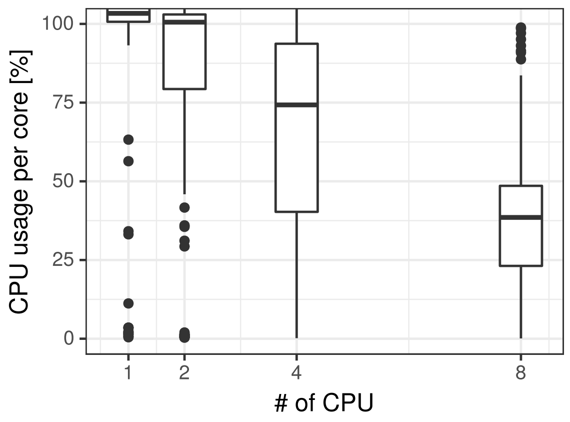


FIGURE 6. Maximum CPU utilization.

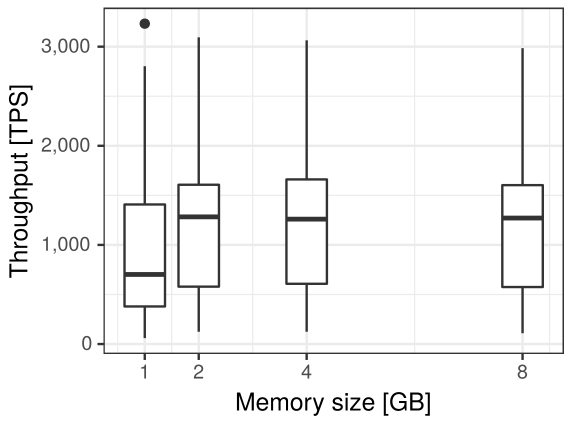


FIGURE 7. Throughput versus memory size (normal transactions, $N_{\text{seal}} = 3$, $N_{\text{CPU}} = 4$, $B = 1$ s).

Throughput was then clarified when normal transactions were processed in the deployed blockchain. FIGURE 5 shows TPS versus the number of CPUs assigned to each sealer. Throughput increased as the number of CPUs increased. However, the growth gradually slowed down. We then see how much CPU was utilized. FIGURE 6 shows CPU utilization measured by `docker stats` after the transactions were sent out. CPU utilization is defined as the maximum CPU utilization divided by the number of assigned CPUs. For example, when CPU utilization is 400% in `docker stats` with a docker container that is assigned eight CPUs, the maximum CPU utilization is 50%(= 400%/8). FIGURE 6 shows that as the number of CPUs increased, the CPU utilization decreased. This result can be explained in two ways. The first is that the workload of `geth` may not have been high enough to use all the CPUs up. The second explanation is that there might have been a disk I/O bottleneck rather than a CPU bottleneck. Similarly, we clarified how memory size affects throughput. FIGURE 7 shows throughput versus assigned memory size. We can see that throughput remained stable except for the case of 1 GB, meaning that memory size did not affect throughput as long as enough memory size, that is, 2 GB in our case, was assigned.

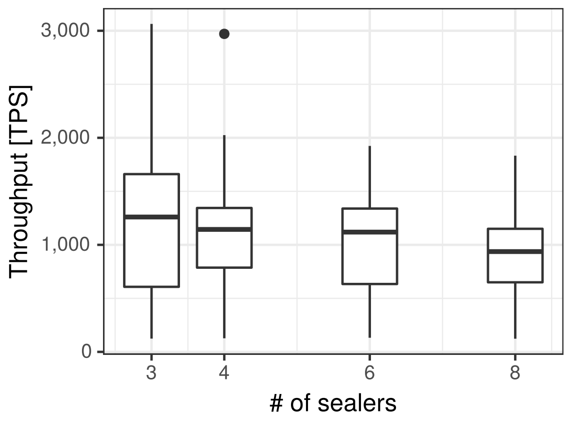


FIGURE 8. Throughput versus the number of sealers (normal transactions, N_{CPU} = 4, M = 4 GB, B = 1 s).

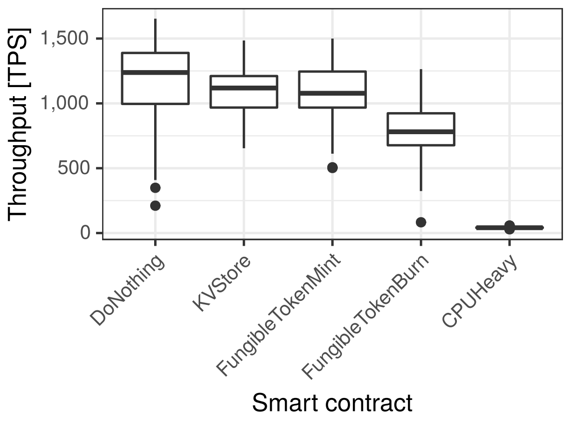


FIGURE 10. Throughput by the smart contracts (N_{seal} = 3, M = 4 GB, B = 1 s).

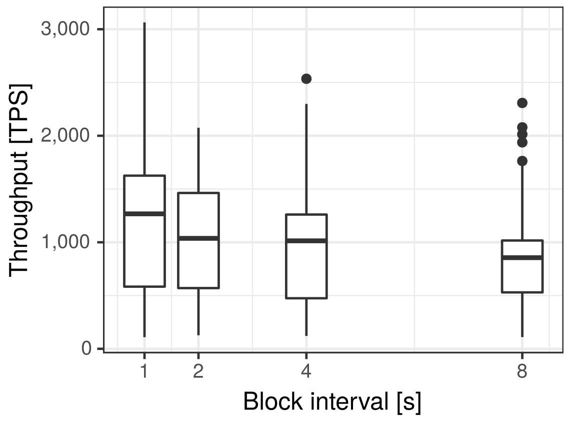


FIGURE 9. Throughput versus block interval (normal transactions, N_{CPU} = 4, M = 4 GB, N_{seal} = 3).

We also clarified how N_{seal} and B affect throughput. FIGURE 8 shows throughput versus the number of sealers. Throughput decreased as the number of sealers increased. For instance, throughput decreased to about 500 TPS when the number of sealers was six. This occurred because sealers should transfer more transactions and blocks among themselves, thus making it more difficult to achieve a consensus. FIGURE 9 shows throughput versus block intervals. As expected, the longer the block interval was, the longer the duration for which the transactions needed to be involved in the blocks. However, this might be true only when sealers are in the same local network, and network latency can be almost ignored.

FIGURE 10 shows the throughput by the smart contracts. As this figure shows, the median value of throughput was 1,250 TPS in `DoNothing`, meaning that the process time was almost same as that of a normal transaction given the same parameter choice. The throughputs of `KVstore` and `FungibleToken`'s mint functions were around 1,150 TPS and lower than that of `DoNothing`. We can also see that the throughput of `FungibleToken`'s burn function was about 770 TPS, much lower than that of the above three functions.

TABLE 7. Top-five bottlenecks and their mean process time measured by `pprof`.

FUNCTION	PROCESS TIME [S]
`runtime.cgocall`	4.4
`runtime.futex`	1.1
`runtime.madvise`	0.87
`runtime.memclrNoHeapPointers`	0.61
`runtime.notetsleep_internal`	0.48

These functions were composed of one operation each, that is, storing, addition and subtraction. We can also see the slight difference among them due to their differing levels of complexity. As `CPUheavy` iterates comparison and storage many times, its median value was around 40.

B. BOTTLENECK ANALYSIS

Given that we clarified the numerical aspects of PoA Ethereum, we then performed the bottleneck analysis to determine the root causes of performance limitation. We first identified the functions that had a significant impact on the sealers' transaction process with `pprof`, which counts the number of function calls and monitors the process time for 30 seconds. TABLE 7 lists the top-five functions that had a significant impact on the process of transactions. As this table shows, `runtime.cgocall` was the most dominant function. Actually, this function was used to call functions in C language from golang. To identify what functions called `runtime.cgocall`, we used `pprof`'s visualization tools. FIGURE 11 shows a cropped call flow with information on `pprof`'s CPU profiling. The process time of functions is also indicated in a box. From this figure, we can see that `runtime.cgocall`, called by `crypto.Ecrecover`, took 5.91 seconds for its calculation.

In addition, we used `pprof`'s *flame graph* representation, shown in FIGURE 12. In a flame graph, function calls are vertically displayed and the horizontal length of functions represents how much time is spent on the functions. As this figure (and the implemented

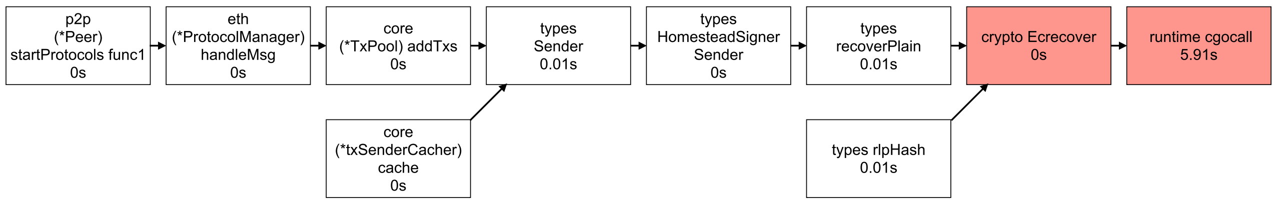

p2p (*Peer) startProtocols func1 0s
eth (*ProtocolManager) handleMsg 0s
core (*TxPool) addTxs 0s
types Sender 0.01s
types HomesteadSigner Sender 0s
types recoverPlain 0.01s
crypto Ecrecover 0s
runtime cgocall 5.91s
core (*txSenderCacher) cache 0s
types rlpHash 0.01s

FIGURE 11. An example of a function call trace by `pprof` **(normal transactions, cropped).**

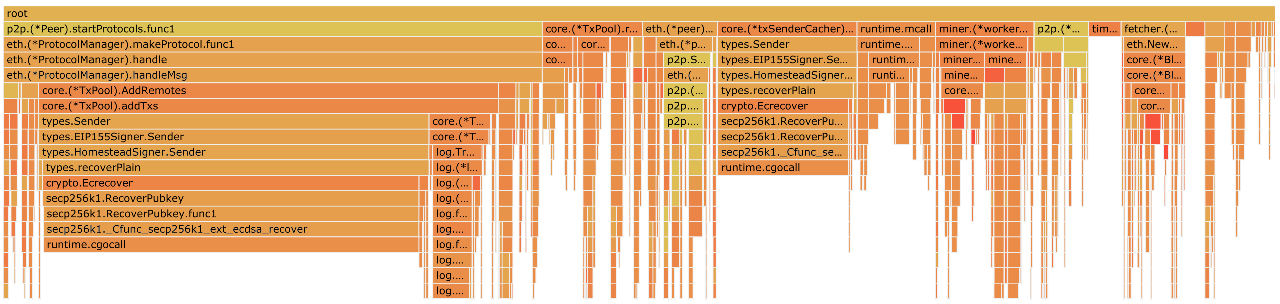


FIGURE 12. An example of a flame graph by `pprof` **(cropped).**

golang file) show, `crypto.Ecrecover` only called `secp256k1.RecoverPubkey`, which recovers a public key from a signature in a transaction. Eventually, `secp256k1.RecoverPubkey` called its C language implementation, `C.secp256k1_ext_ecdsa_recover`.

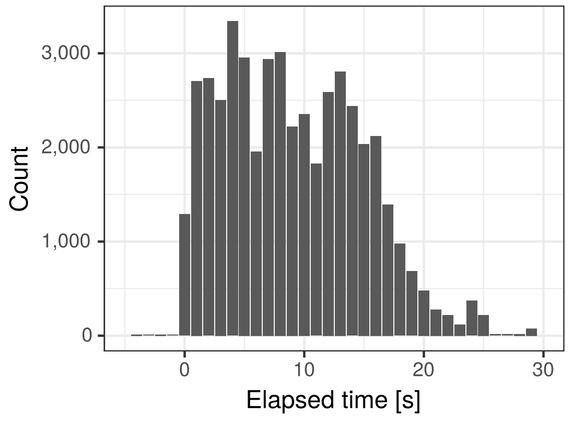


FIGURE 13. The histogram of the `crypto.Ecrecover` **call count after the transactions were sent out.**

FIGURE 13 shows the histogram of the call count of `crypto.Ecrecover` after the transactions were sent out. We used our logging function (FIGURE 2) to measure the number of function calls per second and the average process time of a function. As three sealers were deployed, the result was obtained by averaging each sealer. As FIGURE 13 shows, after the transactions were sent out, that is, elapsed time was $\geq$ 0 s, the number of function calls was suddenly increased, often exceeding 2,500 per second. From the series of log files, the average process time of a `crypto.Ecrecover` was found to be around 400 microseconds. Hence, almost 100% (2, 500/ s $\times$ 400 ms) of the computation was spent on `crypto.Ecrecover` during a thousands of transactions were being processed.

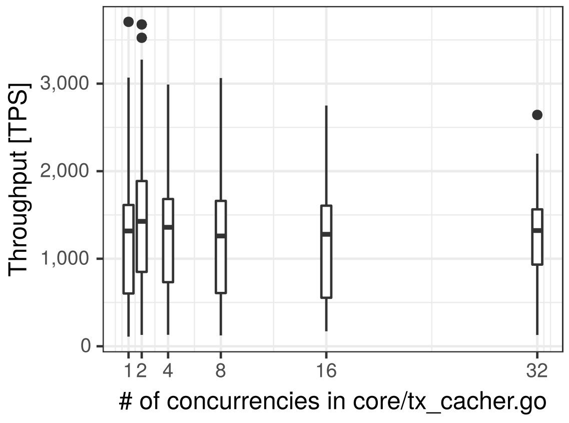


FIGURE 14. Throughout versus the number of concurrencies in `core/tx_cacher.go`.

We analyzed how multi-threading in golang was utilized in `geth`. In golang, goroutine, a lightweight thread managed in golang, is used for threading. The `core.(*txSenderCacher) cache` shown at the top-right corner consists of the concurrent threads dealing with `crypto.Ecrecover`. The number of created threads is equal to the number of CPU cores as defined in `core/tx_cacher.go`. To investigate how the concurrency set in `core/tx_cacher.go` is effective, we varied the number of concurrency from 1 to 32 as the number

TABLE 8. Threats to validity.

VALIDITY	POSSIBLE THREATS
Construct validity	Nil.
Internal validity	The result of CPU usage in FIGURE 6 may not fully reflect on `geth`'s computation workload.
External validity	The validity of the obtained results may not be applicable to other `geth`'s versions or consensus algorithms.
Reliability	The results of performance evaluation may be affected by the specification and running processes of workstations.

of CPU cores of our workstations is 32. FIGURE 14 shows throughput versus the number of concurrencies in `core/tx_cacher.go`. As this figure shows, throughput did not improve even though the number of concurrencies increased. From this result, there is enough room for CPU utilization, and thus, further improvement may be possible by extending the idea of concurrency.

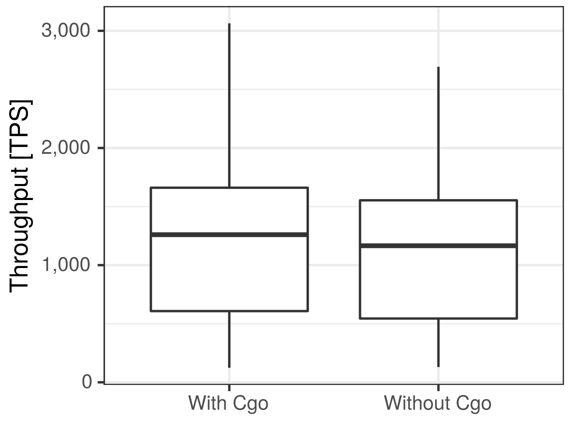


FIGURE 15. Throughput with and without Cgo (normal transactions, $N_{seal} = 3$, $N_{CPU} = 4$, $M = 4$ GB, $B = 1$ s).

Now, we know that the bottleneck was in the process of public key recovery from a signature. Hence, the straightforward approach to fix the bottleneck was not to use the C-implemented `crypto.Ecrecover`. As the native golang version of `crypto.Ecrecover` is also provided in the current `geth`, we evaluated the throughput with and without C functions from golang (Cgo), and the result is shown in FIGURE 15. As this figure shows, throughput was slightly degraded when the native golang version of `crypto.Ecrecover` was used. Hence, it is necessary to explore other ways to improve throughput.

VI. THREATS TO VALIDITY

In order to clarify the validity of our findings, it is important to assess the threats to validity. For this, we refer to Runeson and Höst's guidelines for assessing the threats to validity in software engineering [26]. There are four validity types: (i) construct validity, (ii) internal validity, (iii) external validity and (iv) reliability. TABLE 8 lists the possible threats to validity. We identified three threats that are relevant to internal validity, external validity and reliability.

First, there is a threat to internal validity due to the fact that CPU usage provided in FIGURE 6 may not fully reflect `geth`'s computation workload. When the number of CPUs is four or eight, it is possible that CPUs are not fully utilized since they might be idle during the time spent waiting for I/O.

Second, we identified a threat to external validity. As we only conducted tests with `geth` 1.19.10-stable for private Ethereum, we cannot generalize our findings to other versions of `geth` and public (PoW-based) `geth`. Since our toolset is independent of its versions and consensus algorithms, we can mitigate this threat by testing our methodology in different environments. However, it is necessary to take into account the effect of network latency for evaluating the bottleneck of public Ethereum.

Finally, there is a threat to reliability due to the fact that the measured performance metrics may depend on the specification and other running processes of workstations. More specifically, all the quantitative results, that is, delay of transactions, throughput, CPU usage and memory consumption, may be dependent on the specification of workstations, for example, equipped CPU, memory and disk speed. Since our toolset can be deployed to other workstations, this threat can be mitigated by collecting more results from different workstations. Furthermore, the threat is also affected by the running processes of workstations. We have mitigated this threat by executing the minimum processes in the Docker containers.

VII. CONCLUSION AND FUTURE WORK

We have analyzed the function-level bottleneck of the private PoA Ethereum blockchain by leveraging `pprof`, a golang's profiling tool, and our custom functions. We have clarified that the bottleneck of the current PoA Ethereum is the repeated calculation of the `crypto.Ecrecover` function that extracts a public key from a signature. Although there are two implementations of `crypto.Ecrecover` in PoA Ethereum, — Cgo-based and non-Cgo-based ones —, a slightly better throughput can be achieved via the former. More than 1,000 TPS can be achieved in most smart-contract settings, which is much fewer than 10,000 TPS that are often demanded. However, we have also clarified that much room for improvement remains; the multi-thread functionality by goroutine does not seem to be well utilized in the current `geth`. Hence, we will seek a novel pipelining approach to achieve better throughput in future work.

REFERENCES

[1] ConsenSys. (Apr. 2019). *90+ Ethereum Apps You Can Use Right Now*. Accessed: Feb. 27, 2020. [Online]. Available: https://consensys.net/blog/news/90-ethereum-apps-you-can-use-right-now/

[2] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A framework for analyzing private blockchains," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, 2017, pp. 1085–1100.

[3] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang, ''Untangling blockchain: A data processing view of blockchain systems,'' *IEEE Trans. Knowl. Data Eng.*, vol. 30, no. 7, pp. 1366–1385, Jul. 2018.

[4] S. Rouhani and R. Deters, ''Performance analysis of Ethereum transactions in private blockchain,'' in *Proc. 8th IEEE Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Nov. 2017, pp. 70–74.

[5] G. Wood, ''Ethereum: A secure decentralised generalised transaction ledger,'' Ethereum Project Yellow Paper, 2014, pp. 1–32. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[6] I. Barinov, V. Baranov, and P. Khahulin. (2018). *POA Network Whitepaper*. [Online]. Available: https://github.com/poanetwork/wiki/wiki/POA-Network-Whitepaper

[7] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen, ''Performance analysis of consensus algorithm in private blockchain,'' in *Proc. IEEE Intell. Vehicles Symp. (IV)*, Jun. 2018, pp. 280–285.

[8] M. Bez, G. Fornari, and T. Vardanega, ''The scalability challenge of Ethereum: An initial quantitative analysis,'' in *Proc. IEEE Int. Conf. Service-Oriented Syst. Eng. (SOSE)*, Apr. 2019, pp. 167–176.

[9] M. Schäffer, M. D. Angelo, and G. Salzer, ''Performance and scalability of private Ethereum blockchains,'' in *Proc. Int. Conf. Bus. Process Manage.* Cham, Switzerland: Springer, 2019, pp. 103–118.

[10] J. Al-Jaroodi and N. Mohamed, ''Blockchain in industries: A survey,'' *IEEE Access*, vol. 7, pp. 36500–36515, 2019.

[11] D. Loghin, G. Chen, T. T. A. Dinh, B. C. Ooi, and Y. M. Teo, ''Blockchain goes green? An analysis of blockchain on low-power nodes,'' 2019, *arXiv:1905.06520*. [Online]. Available: http://arxiv.org/abs/1905.06520

[12] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong, ''Performance analysis of private blockchain platforms in varying workloads,'' in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2017, pp. 1–6.

[13] S. K. Kim, Z. Ma, S. Murali, J. Mason, A. Miller, and M. Bailey, ''Measuring Ethereum network peers,'' in *Proc. Internet Meas. Conf.*, Oct. 2018, pp. 91–104.

[14] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, and S. Chatterjee, ''Performance characterization of hyperledger fabric,'' in *Proc. Crypto Valley Conf. Blockchain Technol. (CVCBT)*, Jun. 2018, pp. 65–74.

[15] Q. Nasir, I. A. Qasse, M. A. Talib, and A. B. Nassif, ''Performance analysis of hyperledger fabric platforms,'' *Secur. Commun. Netw.*, vol. 2018, pp. 1–14, Sep. 2018.

[16] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, ''Performance modeling of hyperledger fabric (permissioned blockchain network),'' in *Proc. IEEE 17th Int. Symp. Netw. Comput. Appl. (NCA)*, Nov. 2018, pp. 1–8.

[17] P. Thakkar, S. Nathan, and B. Viswanathan, ''Performance benchmarking and optimizing hyperledger fabric blockchain platform,'' in *Proc. IEEE 26th Int. Symp. Modeling, Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2018, pp. 264–276.

[18] B. Ampel, M. Patton, and H. Chen, ''Performance modeling of hyperledger sawtooth blockchain,'' in *Proc. IEEE Int. Conf. Intell. Secur. Informat. (ISI)*, Jul. 2019, pp. 59–61.

[19] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, ''Dissecting ponzi schemes on Ethereum: Identification, analysis, and impact,'' *Future Gener. Comput. Syst.*, vol. 102, pp. 259–277, Jan. 2020.

[20] W. Chen, Z. Zheng, J. Cui, E. Ngai, P. Zheng, and Y. Zhou, ''Detecting ponzi schemes on Ethereum: Towards healthier blockchain technology,'' in *Proc. World Wide Web Conf. World Wide Web (WWW)*, 2018, pp. 1409–1418.

[21] C. F. Torres and M. Steichen, ''The art of the scam: Demystifying honeypots in Ethereum smart contracts,'' in *Proc. USENIX Secur. Symp.*, 2019, pp. 1591–1607.

[22] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, ''Making smart contracts smarter,'' in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 254–269.

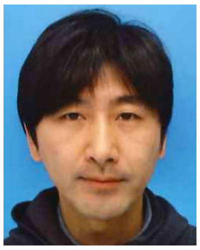
[23] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, ''SmartCheck: Static analysis of Ethereum smart contracts,'' in *Proc. 1st Int. Workshop Emerg. Trends Softw. Eng. Blockchain (WETSEB)*, 2018, pp. 9–16.

[24] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, ''EthIR: A framework for high-level analysis of Ethereum bytecode,'' 2018, *arXiv:1805.07208*. [Online]. Available: http://arxiv.org/abs/1805.07208

[25] J. W. Tukey, *Exploratory Data Analysis*, vol. 2. Reading, MA, USA: Addison-Wesley, 1977. [Online]. Available: https://books.google.com.sg/books/about/Exploratory_Data_Analysis.html?id=UT9dAAAAIAAJ&redir_esc=y

[26] P. Runeson and M. Höst, ''Guidelines for conducting and reporting case study research in software engineering,'' *Empirical Softw. Eng.*, vol. 14, no. 2, p. 131, 2008.

KENTAROH TOYODA (Member, IEEE) was born in Tokyo, Japan, in 1988. He received the B.E., M.E., and Ph.D. degrees in engineering from the Department of Information and Computer Science, Keio University, Yokohama, Japan, in 2011, 2013, and 2016, respectively. He was an Assistant Professor with Keio University, from April 2016 to March 2019, and is currently a Scientist with the Agency for Science, Technology, and Research (A*STAR) Research Entities, Singapore Institute of Manufacturing Technology (SIMTech), Singapore. His research interests include blockchain analysis and its applications, security and privacy, data analysis, Industry 4.0, and eHealth. He received the IEICE Communication Society Encouragement Awards, in 2012 and 2015, the Telecom System Technology Encouragement Award, in 2015, and the Fujiwara Foundation Award of Keio University, in 2016. He is a member of the IEICE.



KOJI MACHI was born in Tokyo, Japan. He received the bachelor's degree in social science from Waseda University, Japan, in 2013. He is the CEO of SingulaNet Ltd., and the Founder of Londobell Labs Pte. Ltd.

YUTAKA OHTAKE was born in Aomori, Japan. He received the bachelor's degree in computer science from The University of Electro-Communications, in 1998. He is the CTO of SingulaNet Ltd.

ALLAN N. ZHANG (Associate Member, IEEE) received the B.Sc., M.Eng., and Ph.D. degrees from Wuhan University, China, in 1986, 1989, and 1992, respectively. He is a Senior Scientist with the Agency for Science, Technology, and Research (A*STAR), Singapore Institute of Manufacturing Technology, Singapore. He has over 20 years of experience in knowledge-based systems and enterprise information systems development. He and his team are currently involved in the research of manufacturing system analyses, including data mining, supply chain information management, supply chain risk management using complex systems approach, multi-objective vehicle routing problems, and urban last-mile logistics. His current research interests include knowledge management, data mining, machine learning, artificial intelligence, computer security, software engineering, software development methodology and standard, and enterprise information systems.