IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received August 25, 2021, accepted September 7, 2021, date of publication September 16, 2021, date of current version October 11, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3113522

Error-Correction Code Proof-of-Work on Ethereum

HYOUNGSUNG KIM[1], **JEHYUK JANG**[1], **SANGJUN PARK**[2], **AND HEUNG-NO LEE**[1], **(Senior Member, IEEE)**
[1]School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Gwangju 61005, South Korea
[2]Electronics and Telecommunications Research Institute (ETRI), Gwangju 500-712, South Korea

Corresponding author: Heung-No Lee (heungno@gist.ac.kr)

This research was supported in part by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2021-0-01835) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), in the part by the IITP Grant through Korean Government MSIT under Grant 2020-0-00958, and in part by the National Research Foundation of Korea (NRF) Grant through Korean Government MSIP under Grant NRF-2021R1A2B5B03002118.

ABSTRACT The error-correction code proof-of-work (ECCPoW) algorithm is based on a low-density parity-check (LDPC) code. ECCPoW can impede the advent of mining application-specific integrated circuits (ASICs) with its time-varying puzzle generation capability. Previous research studies on ECCPoW algorithm have presented its theory and implementation on Bitcoin. In this study, we have not only designed ECCPoW for Ethereum, called ETH-ECC, but have also implemented, simulated, and validated it. In the implementation, we have explained how ECCPoW algorithm has been integrated into Ethereum 1.0 as a new consensus algorithm. Furthermore, we have devised and implemented a new method for controlling the difficulty level in ETH-ECC. In the simulation, we have tested the performance of ETH-ECC using a large number of node tests and demonstrated that the ECCPoW Ethereum works well with automatic difficulty-level change capability in real-world experimental settings. In addition, we discuss how stable the block generation time (BGT) of ETH-ECC is. Specifically, one key issue we intend to investigate is the finiteness of the mean of ETH-ECC BGT. Owing to a time-varying cryptographic puzzle generation system in ECCPoW algorithm, BGT in the algorithm may lead to a long-tailed distribution. Thus, simulation tests have been performed to determine whether BGT distribution is not heavy-tailed and has a finite mean. If the distribution is heavy-tailed, stable transaction confirmation cannot be guaranteed. In the validation, we have presented statistical analysis results based on the two-sample Anderson–Darling test and discussed how the BGT distribution follows an exponential distribution which has a finite mean. Our implementation is available for download at https://github.com/cryptoecc/ETH-ECC.

INDEX TERMS Anderson–Darling test, ASIC-resistant, blockchain, error-correction codes, Ethereum, hypothesis test, LDPC, proof-of-work, simulation, statistical analysis.

I. INTRODUCTION

Blockchain is a peer-to-peer (P2P) network that consists of trustless nodes. In a reliable P2P network, no peers (nodes) would intentionally send wrong information to others. In contrast, in an unreliable P2P network (e.g., a group of trustless nodes), the possibility that some peers may send false information to others should be considered. For example, a node may spread wrong or fake information to others. To address these issues in an unreliable P2P network, Nakamoto proposed using blocks and chaining these blocks with a novel consensus algorithm [1].

In a blockchain, a peer sends a new block containing transactions to other peers. These peers validate the received block and link it to the previous block when there is no problem in the received block, i.e., when the authenticity of the block has been verified. A consensus algorithm is used to accomplish this verification task. If a peer has sent false information to others, such information is detected by the consensus algorithm as there is no collusion among the peers. A generated block contains information about previous blocks, i.e., all blocks are chained; thus, if someone wants to change one block in a chain, all previous blocks of the block to be changed must also be changed. Therefore, unless the network is centralized within a particular group, sending fake information about previous blocks to new peers is impossible.

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

Therefore, to prevent collusion, an unreliable network should avoid centralization.

Nakamoto proposed a proof-of-work (PoW) system for a consensus algorithm. In the PoW system, peers repeat a type of work to solve a cryptographic puzzle using a hash function (e.g., SHA256 [1] and Keccak [2]). When a peer successfully solves a cryptographic puzzle, the peer generates a block. In addition, the peer gets an incentive as a reward for the work done. In an ideal PoW system, new nodes can join to work and receive as much reward as they complete work. However, with an increase in the price of reward, attempts have been made to centralize the network to monopolize incentives.

Centralization is a phenomenon that occurs in PoW-based blockchain networks. In blockchains using PoW as a consensus algorithm, an oligarchy of miners with a disproportionate share of computation resources can monopolize block generation. Such centralization negatively impacts the credibility of a blockchain. For example, in a centralized network, a group of dominant nodes can selectively filter out some transactions belonging to others for their benefit. New nodes will find it difficult to earn trust and join the network in the fear of possible unfair treatment [3], [4].

The emergence of application-specific integrated circuits (ASICs) has accelerated the centralization of PoW. As more nodes use ASICs in generating blocks, the computation complexity in block generation increases. Thus, it has become difficult to generate blocks using general-purpose units, such as a central processing unit (CPU) and a graphics processing unit (GPU). As a result, a few groups equipped with powerful ASICs have surfaced and centralized the blockchain networks. To avoid centralization, researchers have proposed the use of ASIC-resistant PoW (e.g., Ethash of [2], X11 of [12], and Random X of [24]) and alternative consensus algorithms (e.g., proof-of-stake, delegated proof-of-stake, and Byzantium fault tolerance [25]). Networks using alternative algorithms have presented lesser decentralization effects than those have using ASIC-resistant PoW [25]. Specifically, in networks using alternative algorithms, only limited participants can generate blocks, but ASIC-resistant PoW has no limit on the number of participants. Thus, ASIC-resistant PoW presents a more decentralized network than do alternative algorithms.

For an ASIC-resistant PoW, an error-correction code proof-of-work (ECCPoW) algorithm was proposed [6], [7]. In ECCPoW algorithms, a hash value of a previous block generates a varying parity-check matrix (PCM) for error correction. This varying PCM works as a cryptographic puzzle in ECCPoW. These time-varying cryptographic puzzles make ECCPoW ASIC resistant. It is possible to use an ASIC for a specific fixed cryptographic puzzle. In ECCPoW, every newly created puzzle differs from all the previously created puzzles. As a result, if there was an ASIC for ECCPoW, such an ASIC must cover a wide range of cryptographic puzzle generation systems. Such a system, however, would incur huge chip space and cost [10], [11].

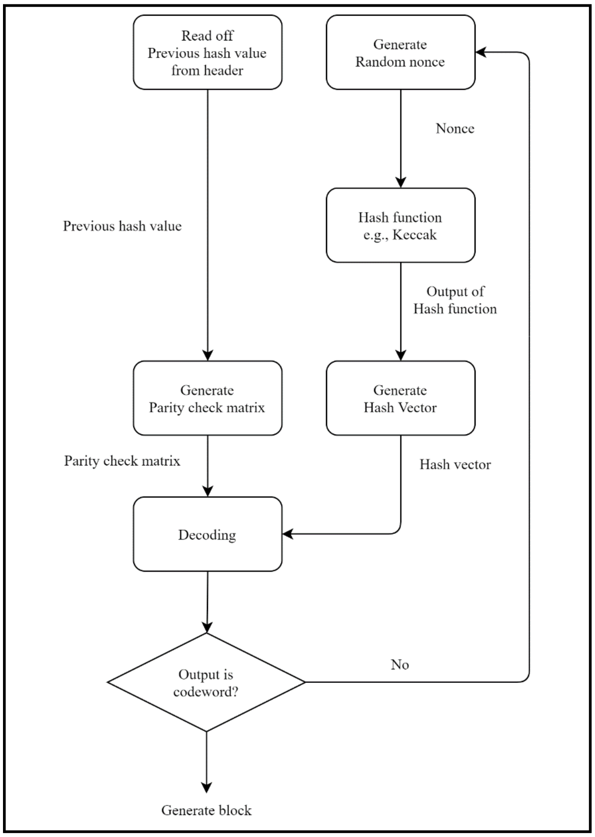


FIGURE 1. **Flowchart of ECCPoW Ethereum. Every miner who generates blocks can construct a parity check matrix using a previous hash value. A generated nonce becomes an input of a hash function. A hash vector used for decoding can be generated using the output of a hash function. If decoding is successful, the block is generated; otherwise, a miner generates a new nonce to make a new hash vector for decoding.**

In [7], the authors have reported that the time-varying puzzle system may generate large block generation time (BGT), i.e., outliers, for ECCPoW implemented on Bitcoin. If outliers occur frequently, it is of our interest to see whether or not the distribution of BGT is heavy-tailed with an infinite mean [15], [26]. As a result, the proposition made in [6] that BGT has a finite mean needs to be challenged. Previous works on ECCPoW [6], [7] did not include sufficient real-world experiments to conclude that BGT has a finite mean. If BGT does not have a finite mean, ECCPoW cannot be used as a consensus algorithm. In this paper, we aim to study the distribution of BGT of ECCPoW implemented on Ethereum (ETH-ECC). Our experimental results show that the BGT distribution is not heavy-tailed and has a finite mean.

The contributions of our work are as follows:

- We show how ECCPoW is implemented on Ethereum.
- We present a method for controlling the difficulty level in ETH-ECC and report the results of automatic difficulty level change with real-world experiments of ETH-ECC.
- We present a goodness-of-fit result using the Anderson–Darling (AD) test for distribution validation and discuss

the BGT distribution follows the exponential distribution which has a finite mean.

The remainder of this paper is organized as follows. Section II provides a background of the requirements of an ASIC-resistant PoW. Section III demonstrates the implementation of ETH-ECC. Section IV discusses the formulation of the problem. Section V provides the experimental result of the implementated ETH-ECC. Finally, Section VI summarizes our work and concludes the paper.

II. BACKGROUND

We introduce three approaches that can be used to avoid centralization problems in PoW. The first is an intentional bottleneck between an arithmetic logic unit (ALU) and memory, which is used by Ethash of Ethereum [2], [5]. It is also termed a memory-hard technique. The second is the *high complexity of ASIC design* used by Dash [12], Raven [13], and our method, ECCPoW. The third is *hybrid methods* of two methods; Random X of Monero uses *hybrid methods* [24].

A. INTENTIONAL BOTTLENECK

The most known PoW of the intentional bottleneck is Ethash of Ethereum [2], [5]. This method uses the difference between the throughput of ALU and the bandwidth of the memory. If there is a bottleneck between the ALU and memory, it is impossible to use the entire throughput of ALU. Specifically, if a miner needs to obtain data from memory to generate a block, the number of block generation attempts is determined by memory bandwidth. Ethash uses a directed acyclic graph (DAG), which is a set of randomly generated data for the bottleneck. The DAG is a huge dataset that cannot be stored in a cache memory; therefore, the DAG is stored in memory. To generate a block using Ethash, a miner must mix a part of the DAG that is stored in the memory. Owing to this procedure, the miner cannot avoid the bottleneck because of limited memory bandwidth. This method has been ASIC resistant for a long time; however, Bitmain released ASIC for Ethash in 2018.

B. HIGH COMPLEXITY OF ASIC DESIGN

Because of the high complexity of ASIC design, ASICs are less efficient. For example, if ASICs are less efficient than a general-purpose unit such as CPU or GPU, there is no reason to design ASIC. X11 of Dash [12] and X16R of Raven [13] use this method. Unlike PoW of Bitcoin, which uses only one hash function (SHA-256), X11 uses 11 hash functions consecutively: BLAKE, BMW, Grosetl, JH, Keccak, Skein, Luffa, Cubehash, SHAvite-3, SIMD, and ECHO. The BLAKE, which is the first hash function of X11, uses a block header with nonce as inputs; its output becomes the input of the next hash function. Similarly, the next hash function uses the output of the previous hash function. This procedure is repeated until a result is obtained for the last hash function. Miners determine whether they have found a valid nonce using the output of the final hash function.

Designing an ASIC for X11 was expensive; therefore, X11 was ASIC resistant. However, Bitmain released an ASIC for X11 in 2016. There are a few PoW algorithms that extend X11 (e.g., X13, X14, and X15); however, the ASICs for these have been released. X16R of Raven is an extended version of X11 of Dash. In X16R, unlike the previous extension of X11, the sequence of 16 hash functions is randomly changed. Therefore, it is costly to design an ASIC for X16R. However, T. Black, who designed X16R, mentioned that there is some evidence that ASICs for X16R exist [23]. Our ECCPoW employs a *time-varying puzzle generation system* to make ASCI design difficult. ECCPoW can make ASIC powerless as the puzzle generation system changes from block to block. We explain this further in Section III.

C. HYBRID METHODS

Random X of Monero combines the above two methods. Random X uses memory-hard techniques for the bottleneck with random code execution; Random X is optimized for CPU mining [24]. In [24], they mentioned that mining can be performed using a field-programmable gate array; however, it will be much less efficient than CPU mining. It implies that efficient mining hardware can be developed when the cost of developing chipsets is low in comparison to the mining reward. With the proposed ECCPoW, attempts in developing efficient mining hardware can be made when the reward-to-cost ratio increases. However, such attempts can be easily evaded since the parameters of ECCPoW can be easily changed, such as increasing the length of code and the code rate. The next section illustrates further the ASCI-resistance characteristic of ECCPoW.

III. ECCPoW IMPLEMENTED ON ETHEREUM

In this section, we briefly introduce ECCPoW and present how ECCPoW has been implemented on Ethereum using Fig. 1. Furthermore, we present how the difficulty level of ETH-ECC is automatically controlled.

A. OVERVIEW OF ECCPoW

In a blockchain employing the PoW consensus algorithm, a node solves cryptographic puzzles to publish a block. For a given puzzle, the node who solves the puzzle first obtains the authority to publish a block. For example, in the PoW of Bitcoin, the first node that finds a specific output of the secure hash algorithm (SHA) obtains the authority to publish a block. The PoW of Ethereum uses Keccak instead of SHA. The ECCPoW algorithm proposed in [6] is a PoW consensus algorithm that uses error-correction code, which comprises the low-density parity-check (LDPC) code [8], as a cryptographic puzzle. The ECCPoW algorithm consists of a pseudo-random puzzle generator (PRPG) and an ECC puzzle solver. Fig. 1 presents the flowchart of the ECCPoW algorithm. For every block, the PRPG generates a new pseudo-random LDPC matrix. A new LDPC matrix is distinct from the other previously generated matrices. Such a pseudo-random LDPC matrix takes the role of issuing an independently announced cryptographic puzzle. The ECC puzzle solver uses the LDPC decoder to solve the given announced puzzle. Specifically, to publish a block, a node is required to run through an input header until the LDPC decoder hits a satisfying result; for instance, the output of the decoder is an LDPC codeword (with a certain Hamming weight). In the next subsection, we will discuss ECCPoW implementation on Ethereum with the flowchart presented in Fig. 1.

B. COMPARISON OF ETHASH AND ECCPoW

Ethereum uses `Ethash` for ASIC resistance, and `ETH-ECC` uses ECCPoW for ASIC resistance. In this subsection, we present how `Ethash` and `ETH-ECC` apply ASIC-resistance property to PoW with pseudo-codes.

`Ethash` uses a DAG for ASIC resistance. The DAG is a large size of data and is typically stored in a random access memory (RAM), not in cache memory. It implies that a miner must access the RAM to get the DAG data. Although the miner could be equipped with a high-throughput ALU, the bandwidth access from the RAM to the ALU is limited. That is, the bottleneck is the limited bandwidth of reading DAG information from the RAM; thus, any fast ALU, e.g., an ASIC implementation of `keccak512`, exceeding this bottleneck is of no use. This makes Ethash ASIC resistant.

When a miner reads DAG data from the RAM, the location where the data are read varies. The location of data reading is selected by the "mix"; the mix is a 128-byte hash value generated by the block header and a nonce. The mix is updated using the Fowler-Noll-Vo (FNV) hash function. The miner repeats this process 64 times. After updating the mix, the miner compresses the mix; for compression, the FNV hash is used again. The miner returns a hash value of the result of concatenating the compressed mix and the seed.

If this hash value is less than the desired target, the nonce is validated, and a new block is linked to the previous block. **Algorithm 1** denotes the pseudo-code of `Ethash`.

```
Algorithm 1 Ethash
Require: block header (BH), nonce, DAG
Initialize seed: seed = keccak512(BH, nonce)
Initialize 128 bytes mix:
    mix = concatenate(seed, seed)
for i = 0, 1, 2, ..., 63:
    Get data from DAG using mix:
    data = DAG_lookup(DAG, mix, i)
    update mix: mix = FNV_hash(mix, data)
end for
for i = 0, 4, 8, ..., length(mix):
    Compress mix: cmix = compress_mix (mix, i)
end for
return keccak256 (concatenate(seed, cmix))
```

`Ethash` uses *the intentional bottleneck* for ASIC resistance, but `ETH-ECC` aims to use *a time-varying puzzle generation system* for ASIC resistance. In `ETH-ECC`, two factors make the design of ASICs very difficult. One is flexible code lengths and randomly generated PCMs. The `ECC_puzzle_solver` generates a hash vector of length-n (subsection C) using a nonce; this n determines the code length. The development of an ASIC for a PCM with length n cannot be realized, as the `ETH-ECC` network changes n and the PCM from one block to another block. The `PRPG` creates a PCM $\mathbf{H}$. A PCM uses a BH as a seed; thus, it is randomly generated. All miners that work to extend the same previous block use the same PCM to solve the ECCPoW puzzle. Thus, it is highly expensive, if not impossible, to implement an ASIC that can handle a time-varying PCM [10], [11]. After generating a hash vector and a PCM, a miner works out how to generate an output word. If this output word satisfies a specific condition, the miner is successful at completing ECCPoW; e.g., the output word can be a codeword, and then, a new block is linked to the previous block. **Algorithm 2** denotes the pseudo-code of `ETH-ECC`. In our implementation, we have replaced `Ethash` and all its relevant peripheral systems with ECCPoW; thus, it has the same requirement as `Ethash` except for the DAG. We present more details about `ETH-ECC` in the following subsections.

```
Algorithm 2 ETH-ECC
Require: block header (BH), nonce
Generate hash vector:
    hash_vector = ECC_puzzle_solver(nonce)
Generate parity check matrix: PCM = PRPG (header)
output_word = decoder(PCM, hash_vector)
return output_word
```

C. ECCPoW ON ETHEREUM

In this subsection, we present how the error-correction process is applied to ETH-ECC using Fig. 1.

$$C := \{\mathbf{c}|\mathbf{Hc} = \mathbf{0} \cap \mathbf{c} \in \{0, 1\}^{n\times 1}\} \tag{1}$$

when a PCM $\mathbf{H}$ is given, a code $\mathbf{c}$, satisfying (1), is referred to as an LDPC code. The goal of the ECCPoW algorithm is to find an LDPC code $\mathbf{c}$ using the PCM $\mathbf{H}$, which is derived by PRPG, and a hash vector $\mathbf{r}$, which is obtained using the ECC puzzle solver. For the PRPG, we employ the previous hash value; the previous hash value, known as the parent hash in the Ethereum block header, randomly generates a PCM. Specifically, we use Gallagher's method to create random PCM [9]; we use the previous hash value as a seed of randomness. Thus, PCMs are changed for every block; because every node has the same seed, they use the same PCM until a block is generated [6].

1) ECC PUZZLE SOLVER ON ECCPoW ETHEREUM

Here, we introduce the ECC puzzle solver process in ETH-ECC. Our definitions are based on [6]. The equations below follow the right-hand side of Fig. 1.

Definition 1 (ECC Puzzle Solver): Hash vector **r** in which the size of n can be obtained as follows:

$$s_1 := Keccak(nonce) \in \{0,1\}^{256} \tag{2}$$

where *Keccak* denotes the hash function used in Ethash of Ethereum [5]. We generate a *nonce* in the same way that Ethereum does. Furthermore, for a longer length of a hash vector, we use $s_u := Keccak(s_1) \in \{0,1\}^{256}$ with $u = 2, 3, \ldots, l+1$. We slice or concatenate the result of *Keccak* to generate a flexible length hash vector **r**:

$$\mathbf{r} := \begin{cases} s_1[1:n] & \text{if } n \leq 256 \\ [s_1 \cdots s_l s_{l+1}[1:j]] & \text{if } n > 256 \end{cases} \tag{3}$$

where $l = \lfloor n/256 \rfloor$ and $j = n - 256 \times l$. For example, when n is less than 256, **r** obtains the same length as n, whereas when n is not less than 256, **r** concatenates the results of *Keccak*. This flexible length hash vector is used for ASIC resistance.

2) PoW OF THE LDPC DECODER

The goal of the LDPC decoder is to find a hash vector **c** that satisfies **Hc**= 0. The definition below explains the decoding presented in Fig. 1.

Definition 2 (Decoder): Given a PCM **H**, which is the size of $m \times n$, and hash vector **r**, which is the size of n, are given, the LDPC decoder uses **H** and **r** as inputs and obtains output **c** using the message-passing algorithm [6], [14]. When **c** satisfies (1), **c** becomes an LDPC code, and a miner completes LDPC decoding.

$$D_{np} : \{\mathbf{r}, \mathbf{H}\} \mapsto \mathbf{c} \in \{0,1\}^{n \times 1} \tag{4}$$

A PCM **H** is randomly generated; however, all miners use the same previous hash value, which is derived from the previous block. Therefore, predicting the next PCM to mine a block in advance is impossible. In the PoW of Ethereum, miners change a nonce when they obtain a wrong output. We follow the same procedure as Ethereum to obtain a hash value from *Keccak* with a *nonce*, but ETH-ECC uses one more step (3) to generate a hash vector for decoding. When the code derived by (4) does not satisfy (1), the miner generates a new *nonce* and repeats all steps.

Our method is based on the *high complexity of ASIC design* in Section II for an ASIC-resistant PoW. However, unlike the mentioned method in Section II, ECCPoW generates varying cryptographic puzzles of *high complexity*. Specifically, ECCPoW uses two factors to achieve *high complexity*: flexible length LDPC code **c** and randomly generated PCM **H**. ASICs can be released for the n length of code. However, extending the length of code (e.g., $n + 1$) makes ASICs powerless. Furthermore, in [10], [11], it has been proven that implementing an ASIC that can handle variable PCMs is expensive and occupies a lot of space. If developing an ASIC costs more than buying a CPU or GPU, there is no incentive to develop an ASIC. In other words, the ECCPoW algorithm is ASIC resistant as implementing an ASIC that can handle various lengths of changing codes and randomly generated PCMs is inefficient.

D. DIFFICULTY-LEVEL CONTROL OF ETH-ECC

In this subsection, we demonstrate the implementation of ETH-ECC's difficulty-level control. Bitcoin [1] and Ethereum [2] have different difficulty-level control methods. Furthermore, we present one way to add fine difficulty control.

In Bitcoin, the Bitcoin network changes the difficulty level every 2016 block; the desired BGT is 10 min for a block. If miners generate a block every 10 min, generating 2016 blocks takes precisely 2 weeks. Thus, if generating 2016 blocks takes more than 2 weeks, the difficulty level decreases; otherwise, the difficulty level increases. Unlike Bitcoin, the Ethereum network changes the difficulty level every block. Ethereum network allows for a block to be generated between 9 and 18 s. If a block is generated within 9 s, then the difficulty level increases. If it exceeds 18 s, then the difficulty level decreases. Because of this difference between Bitcoin and Ethereum, ECCPoW-based Bitcoin (BIT-ECC) and ETH-ECC also have different difficulty-level control methods. Thus, ETH-ECC cannot use BIT-ECC's method. Because of the need for a new method, we demonstrate the implementation of ETH-ECC's difficulty level control with a difference from Ethereum's method.

Ethereum uses the number of attempts to generate a block per second, termed hash rate, and a probability of block generation. Similarly, ETH-ECC uses the hash rate but considers a probability of decoding success. In [5], the difficulty of Ethereum is defined by the probability of block generation. The difficulty is as follows:

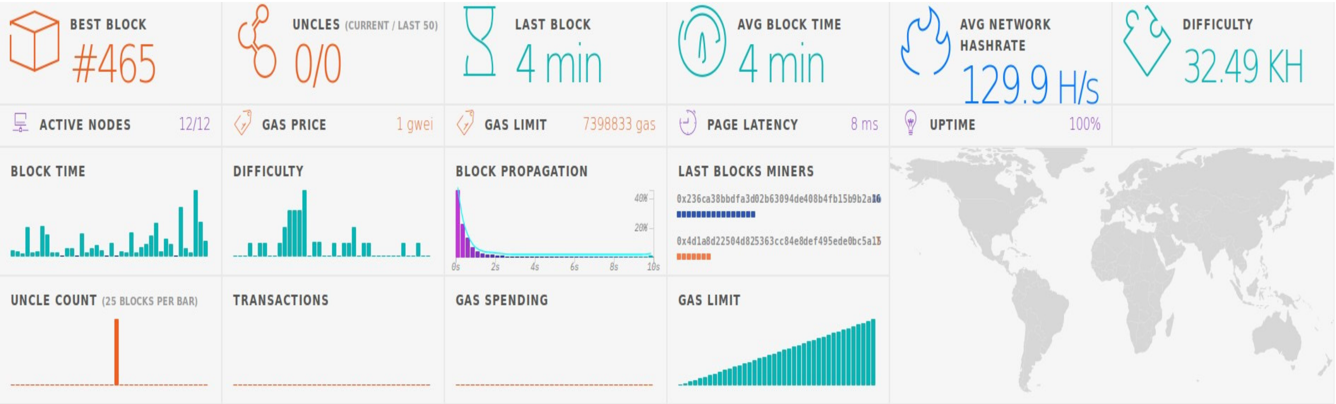
$$n \leq \frac{2^{256}}{Diff} \tag{5}$$

It indicates that

$$Diff \leq \frac{2^{256}}{n} \tag{6}$$

where n denotes the result of PoW and *Diff* denotes the difficulty of Ethereum. Thus, (6) means that when the difficulty level increases, the number of n that satisfies (6) decreases. Furthermore, we can consider that the reciprocal of difficulty is a probability of block generation. Ethereum uses this probability and hash rate to control BGT. For example, without replacement, when the probability of block generation is 1/150 and hash rate is 10 hash per second, brute force takes 15 s. If the hash rate increases, such as 20 hash per second, Ethereum's method adjusts the probability of BGT to 1/300. Thus, brute force takes 15 s even though the hash rate increases.

For ECCPoW, if we can calculate a probability of decoding success, it is possible to control the difficulty level similar to the process in Ethereum. Thus, it is important to know the probability of a successful LDPC decoding according to the LDPC parameter. We use the pseudo-probability of a successful LDCP decoding according to the parameters to test the difficulty level change using the BGT [7]. That is, ETH-ECC uses the probability of decoding success and



BEST BLOCK #465

UNCLES (CURRENT / LAST 50) 0/0

LAST BLOCK 4 min

AVG BLOCK TIME 4 min

AVG NETWORK HASHRATE 129.9 H/s

DIFFICULTY 32.49 KH

ACTIVE NODES 12/12

GAS PRICE 1 gwei

GAS LIMIT 7398833 gas

PAGE LATENCY 8 ms

UPTIME 100%

BLOCK TIME

DIFFICULTY

BLOCK PROPAGATION

LAST BLOCKS MINERS

0x236ca38bbdfa3d02b63094de408b4fb15b9b2a16

0x4d1a8d22504d825363cc84e8def495ede0bc5a15

UNCLE COUNT (25 BLOCKS PER BAR)

TRANSACTIONS

GAS SPENDING

GAS LIMIT

FIGURE 2. This figure shows the simulation results of ECCPoW Ethereum on Amazon Web Services AWS). Twelv nodes are used in the simulation. The tw nodes are bootnodes that help connect th nodes, and the other 10 nodes are sealnodes that participate i block generation. We use the m5.xlarge of AWS EC2 for the simulation. In the charts, BLOCK TIME shows the block generation times for the last 40 blocks, and DIFFICULTY shows the difficulty levels of the last 40 blocks. BLOCK PROPAGATION shows th percentage of the block propagation time corresponding to time.

hash rate to control the difficulty level. For example, without replacement, when the probability of decoding success is 1/150 and the hash rate is 10 hash per second, it takes 15 s, as in the above example of Ethereum's method. However, unlike Ethereum, when the hash rate increase, ETH-ECC tunes parameters of LDPC to adjust the probability of decoding success. By tuning parameters, ECCPoW achieves both difficulty-level control and ASIC resistance. These parameters can be found at https://github.com/cryptoecc/ETH-ECC/blob/master/consensus/eccpow/LDPCDifficulty_utils.go#L65. In Fig. 2, the difficulty of ETH-ECC is 32.49 KH, indicating that the probability of block generation is 1 of 32,490 hash.

One Way to Add Fine Difficulty Control. ECCPoW controls difficulty using integer and discrete variable *n*. Thus, it may look inappropriate to manage difficulty precisely. However, as the number of blocks increase, block generation time (BGT) converges to the ideal BGT time, which is suitable for a network. For example, when there exist two difficulties: n and $n+1$, we can define average BGT of each difficulty as t_n and t_{n+1}. Thus, we can define the average BGT:

$$averageBGT = \frac{\alpha t_n + \beta t_{n+1}}{k} \tag{7}$$

where α denotes the number of generated blocks with difficulty t_n, β denotes the number of generated blocks with difficulty t_{n+1}, and k denotes the total number of generated blocks (TNGB). Thus, α can be replaced as $\alpha = k - \beta$. As a result, equation (7) is:

$$averageBGT = \frac{(k-\beta)t_n + \beta t_{n+1}}{k} \tag{8}$$

When TNGB k is kept constant, the average BGT is determined by the number of generated blocks β in Eq. (8). Thus, the ideal average BGT, which is suitable for the number of nodes in a network, depends on β. In other words, when TNGB $k$is low, the average BGT cannot meet the ideal average BGT because there are not enough blocks of each difficulty. However, as TNGB k increases, the number of blocks corresponding to the difficulty, such as β, getting closer to the proportion that fits the probability of block generation. As a result, average BGT converges to the ideal average BGT; this convergence confirms our proposition that the network can control difficulty precisely.

IV. PROBLEM FORMULATION

In PoW, there is a case that nodes generate blocks at the same time. Bitcoin allows only one block to be generated at a time; Ethereum allows three blocks to generate at the same time. However, in Ethereum, only one block can be canonical; the other blocks cannot. Blocks that cannot be canonical are called uncle blocks. In Ethereum, nodes rollback transactions of uncle blocks [5]. Therefore, the transaction's participants must wait for block confirmation to prevent a rollback. That is, in the blockchain using PoW, the BGT must have a finite mean for the block confirmation time. For example, if the BGT has an infinite mean, the waiting time for the confirmation of transactions cannot be determined. Therefore, to apply the ECCPoW algorithm in a real network, the BGT must have a finite mean.

In [6], the authors presented the definition of the block generation of the ECCPoW algorithm using a hash rate with a geometric distribution. That is, they assumed that nodes generate a block with specific block generation attempts. However, if the BGT has an infinite mean, there is no guarantee that nodes generate a block with specific attempts. In [7], the authors presented a practical experiment using the ECCPoW algorithm. However, they only mentioned that the BGT of ECCPoW is "unstable." That is, they mentioned that the BGT of ECCPoW has outliers; however, they did not present a discussion on the BGT. Thus, in this study, we present a discussion on the BGT. Specifically, our experimental result presents evidence that the exponential distribution describes the distribution of the BGT of ECCPoW.

V. EXPERIMENT ON ETH-ECC

In this section, we conduct experiments using ETH-ECC. First, we simulate the difficulty level change using multinode networks. Second, we conduct a goodness-of-fit experiment using the AD test [16]–[18] to discuss the distribution of the BGT with a fixed difficulty level.

A. SIMULATION OF THE DIFFICULTY CHANGE

We simulate the difficulty-level change employing Amazon Web Services (AWS) using 12 nodes. Two nodes are *bootnodes* that help connect the nodes, and the other 10 nodes are *sealnodes* that participate in block generation. In the charts presented in Fig. 2, *BLOCK TIME* presents the BGT of the last 40 blocks, and *DIFFICULTY* shows the difficulty level of the last 40 generated blocks. *BLOCK TIME* and *DIFFICULTY* show that because of the large standard deviation, a block is gradually generated despite the low difficulty level, as mentioned in [7]; in the next subsection, we discuss the BGT. In the charts presented in Fig. 2, *LAST BLOCK* shows the BGT of the previous block, and *AVG BLOCK TIME* shows the average of the BGT. In addition, *AVG NETWORK HASHRATE* shows the average hash rate of all miners. *BLOCK PROPAGATION* shows the block propagation time from a miner who generated a block to other miners. We used two different regions: Seoul and US East for *sealnodes*. Specifically, 3 of the 10 *sealnodes* are in the US East region, whereas the rest are in the Seoul region. *BLOCK PROPAGATION* also shows the percentage of blocks that are propagated at corresponding times. *BLOCK PROPAGATION* indicates that the propagation of approximately all blocks between Seoul and US East regions takes less than 2 s. The block propagation is the same method as that of Ethereum.

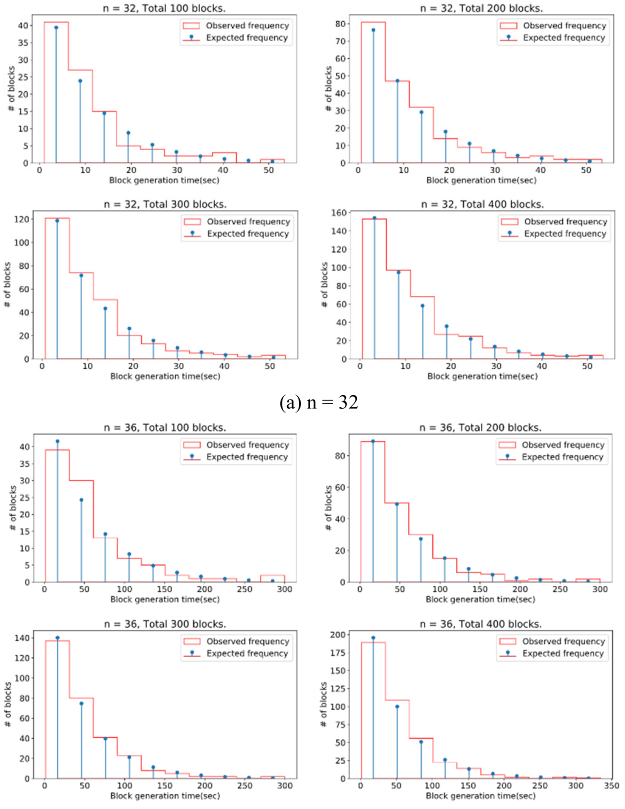
B. STABILITY OF THE BLOCK GENERATION TIME

Fig. 2 demonstrates the importance of determining whether varying puzzles may result in outliers. That is, in *BLOCK TIME* and *DIFFICULTY* of Fig. 2, slow block generations are observed despite the low difficulty level. In other words, the observation of BGT shows outliers. If the outliers are uncontrollable, the BGT distribution has an infinite mean similar to the heavy-tailed distribution. An infinite mean cannot guarantee transaction confirmation. Thus, to achieve a stable BGT that can guarantee transaction confirmation, the BGT must have a finite mean.

We obtain the BGT of ECCPoW Ethereum with a fixed difficulty level to observe the type of distribution with a finite mean the BGT follows. Specifically, if BGT follows an exponential distribution, it has a finite mean. However, if the BGT follows a heavy-tailed distribution, it has an infinite mean [15]. Thus, through the goodness-of-fit experiment, we aimed to discuss what type of distribution the BGT follows. For the goodness-of-fit experiment, we set a null hypothesis H_0 and an alternative hypothesis H_A:

H_0: BGT has the exponential distribution

H_A: BGT does not have the exponential distribution



(a) n = 32

(b) n = 36

FIGURE 3. We did experiments for 100, 200, 300, and 400 blocks to observe the distribution over the number of blocks. As the number of blocks increases, the standard error decrease. That is, when the number of blocks increases, sample distribution reflects an actual distribution of sample distribution. In these figures, experiment results show the tendency; the distributions of observed frequency, known as sample distribution, follow the distribution of expected frequency.

For the goodness-of-fit experiment, we use the AD test [16]–[18]. Other available tests can be used in the goodness-of-fit experiment, such as the chi-square test [19], Kolmogorov–Smirnov test [20], and AD test [16]. The chi-square test has a restrictive assumption that all expected frequencies should be greater or equal to 5 [21]. However, there is no guarantee that our samples will achieve this assumption. If we collect more samples, the chi-square test can be used. However, the *p*-values used to validate the hypotheses are affected by the number of samples. When the number of samples increased in the chi-square test, the *p*-values tend to decrease. Therefore, the assumption of the chi-square test is inappropriate for verifying our distributions. The Kolmogorov–Smirnov test is unaffected by sample sizes; however, it is more sensitive to the center of the distribution rather than the tail [22]. We must consider verifying the tail of the distribution to cover all possibilities. Therefore, we have chosen to use the AD test [16], which gives more weight to the tail than does the Kolmogorov–Smirnov test.

C. AD TESTS

In this subsection, we discuss the AD test and verify its usage using test examples. The AD test is used to verify if a sample

TABLE 1. Example of the Anderson–Darling test results.

The number of samples	Standardized A^2_{MN}	p-value
10	-0.59	$p \geq 0.25$
20	0.44	$p = 0.21$
30	0.69	$p = 0.17$

(a) F $\sim$ Exp(1) , G $\sim$ Normal(1, 1)

10	1.20	$p = 0.11$
20	3.57	$p = 0.02$
30	4.67	$p = 0.01$

(b) F $\sim$ Exp(1) , G $\sim$ Exp(2)

10	1.11	$p = 0.12$
20	-0.41	$p \geq 0.25$
30	-0.08	$p \geq 0.25$

(c) F $\sim$ Exp(1) , G $\sim$ Exp(1)

follows a specific distribution. We discuss one-sample and two-sample AD tests. In our work, we use the two-sample AD test; however, to clearly present our contribution, we briefly introduce the one-sample AD test first.

1) ONE-SAMPLE AD TEST

The one-sample AD test is suitable to verify a hypothesis that a sample set comes from a population. The one-sample AD test is as follows. When the cumulative distribution function (CDF) of the population distribution is $F(x)$, and the CDF of the empirical distribution is $F_M(x)$, the one-sample AD test [18] is used as follows:

$$A^2_M = M \int_{-\infty}^{\infty} (F_M(x) - F(x))^2 \, w(x) dF(x) \tag{9}$$

and

$$w(x) = [F(x)(1 - F(x))]^{-1} \tag{10}$$

where M denotes the number of samples and A^2_M denotes the results of the one-sample AD test. Intuitively, in (9), if $F_M(x) - F(x)$ is 0 for all x, A^2_M is 0. This means that when A^2_M is small, the empirical distribution $F_M(x)$ is close to the population distribution $F(x)$. As we have noted, we focus on the tail of the distribution; it can be accomplished using (10). The one-sample AD test result A^2_M can be used to verify if a given sample comes from a population with a specific distribution.

2) TWO-SAMPLE AD TEST

In our work, we want to verify that two-sample sets come from the same unknown population. The two-sample AD test is appropriate for such verification. The two-sample AD test [17], [18] is as follows. There are two-sample empirical distributions $F_M(x)$ and $G_N(x)$. The $F_M(x)$ is an empirical distribution derived from the set $\mathcal{F}$ with a cardinality of the sample set $M = |\mathcal{F}|$. The $G_N(x)$ is also an empirical distribution derived from the set $\mathcal{G}$ with a cardinality of the sample set $N = |\mathcal{G}|$. $F_M(x)$ and $G_N(x)$ are the respective sample sets independently obtained from two different testing locations. The two-sample AD test can be used to determine whether both sample distributions come from the same distribution. In [17], [18], the two-sample version is defined as follows:

$$A^2_{MN} = \frac{MN}{K} \int_{-\infty}^{\infty} \frac{(F_M(x) - G_N(x))^2}{H_K(x)(1 - H_K(x))} dH_K(x) \tag{11}$$

where $H_K(x) = (MF_M(x) + NG_N(x))/K$ with $K = M + N$. A^2_{MN} is standardized to remove the dependencies derived by the number of samples. This standardized form is used to calculate the p-value [17], [18]. The p-value evidences the hypothesis test.

The two-sample AD test is suitable to verify a hypothesis that two-sample sets come from the same population. As a null hypothesis H_0 for the two-sample AD test, we set $F_M(x)$ to have the same population as $G_N(x)$. In addition, we set $G_N(x)$ as an exponential distribution. Thus, if $F_M(x)$ and $G_N(x)$ comes from same the population, that is, H_0 is true, we may consider that $F_M(x)$ is the exponential distribution. If the p-value of the AD test is sufficiently large, it proves that H_0 is true.

TABLE 2. The observed frequency is calculated using the histogram in Fig. 4, and the expected frequency is calculated using the CDF of the exponential distribution derived from the mean in Fig. 4.

Interval(%)	Observed frequency	Expected frequency
[0, 10)	107	118.70
[10, 20)	82	71.73
[20, 30)	56	43.35
[30, 40)	20	26.19
[40, 50)	14	15.83
[50, 60)	7	9.56
[60, 70)	5	5.78
[70, 80)	4	3.49
[80, 90)	2	2.11
[90, 100]	3	1.27

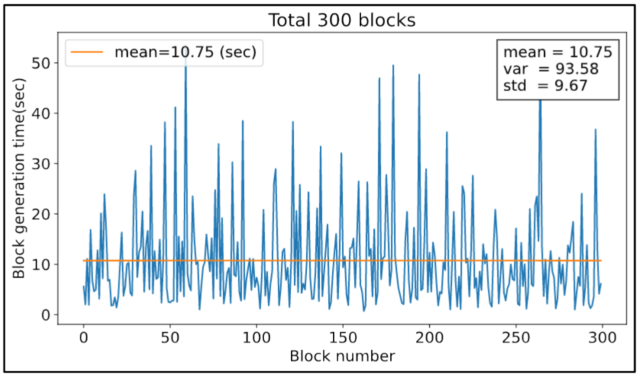


FIGURE 4. This figure presents block generation time of 300 blocks when *n* is 32. The mean block generation time of 300 blocks is 10.75 s, and it is presented as a horizontal line. Such a result is converted to a histogram. The observed frequency of Table 2 denotes the histogram of Fig. 4. The legend at the top right shows the mean, variance, and standard deviation of the BGT.

The p-value is the false positive probability under the assumption that the null hypothesis is true. A low p-value indicates that a test result provides evidence against the null hypothesis; a large p-value does not. That is, a large p-value denotes the probability of a true negative is low. The p-value is determined from the observation of the sample data. Thus, before observing the data, we first set the threshold significance level (*TSL*), $TSL \in [0, 1]$. The *TSL* can be used to determine the critical value. Given a *TSL* and the number of samples that are used in the AD test, the *TSL* table in [18] is used to read off a value corresponding to the TSL and the number of samples. This read-off value is called the critical value. If the standardized A^2_{MN} is smaller than the critical value, this result indicates that the p-value is larger than the predefined *TSL*. In the *TSL* table of [18], the maximum *TSL* is 0.25. Thus, when standardized A^2_{MN} is less than the critical value corresponding to the 0.25 *TSL*, the p-value is capped at 0.25.

3) VERIFICATION OF THE AD TEST

In this subsection, we verify the two-sample AD test method. Verification is performed under the assumption that the input distributions are *a priori* known. This will clearly illustrate how we will use the AD test and interpret its test results.

In Table 1, we present three examples to give an insight into the p-value of the AD test; in this example, we use true distributions for $F_M(x)$ and $G_N(x)$. In Table 1, Exp(θ) indicates the exponential distribution with mean θ and Normal(μ, σ) indicates the normal distribution with mean μ and standard deviation σ. That is, $\mathcal{F} \sim$ Exp(θ) denotes the sample set $\mathcal{F}$ of $F_M(x)$; samples are derived from the exponential distribution with mean θ. In Table 1 (a), we use the exponential distribution for $F_M(x)$ and the normal distribution for $G_N(x)$; these distributions have the same mean. This example shows that as the number of samples increases, the p-value tends to decrease if samples are drawn from different distributions. In Table 1 (b), we set both $F_M(x)$ and $G_N(x)$ as the exponential distribution, but each with different mean values. This example shows that as the number of samples increases even though samples are drawn from the same exponential distribution, the p-value tends to decrease if the means of distributions are different. In Table 1 (c), we set both $F_M(x)$ and $G_N(x)$ to be exactly the same exponential distribution. That is, the two-sample sets $\mathcal{F} \sim F_M(x)$ and $\mathcal{G} \sim G_N(x)$ come from the same population. This example shows that, as the number of samples increases, the p-value tends to increase when two-sample sets are drawn from the same population. From these examples in Table 1, we note that the closer the two distributions $F_M(x)$ and $G_N(x)$ are to each other, the larger p-value is obtained.

We determine whether the AD test result of our experiments indicates that $F_M(x)$ is sufficiently close to $G_N(x)$. That is, given there are two-sample sets, one of $F_M(x)$ and the other of the exponential $G_N(x)$, we want to determine whether we can make a quality statement about how close the two-sample sets are to each other according to the AD test.

TABLE 3. Anderson-Darling test result. The test result presents a large *p*-value. It means that if we reject the null hypothesis, the probability of a true negative is low.

n	# of blocks	Observed mean(sec)	std	Standardized A^2_{MN}	p-value
32	100	10.86	9.84	-1.12	$p \geq 0.25$
32	200	11.24	10.16	-1.20	$p \geq 0.25$
32	300	10.74	9.67	-1.18	$p \geq 0.25$
32	400	11.08	9.84	-1.09	$p \geq 0.25$
32	500	10.91	9.62	-1.11	$p \geq 0.25$
32	600	10.87	9.48	-0.80	$p \geq 0.25$
32	700	10.84	9.41	-0.36	$p \geq 0.25$
32	800	10.76	9.40	-0.36	$p \geq 0.25$
36	100	56.00	55.20	-1.11	$p \geq 0.25$
36	200	51.04	49.71	-1.19	$p \geq 0.25$
36	300	47.84	45.49	-1.12	$p \geq 0.25$
36	400	49.97	47.80	-1.19	$p \geq 0.25$
36	500	49.24	46.95	-1.11	$p \geq 0.25$
36	600	48.23	46.96	-1.18	$p \geq 0.25$
36	700	48.36	47.68	-1.18	$p \geq 0.25$
36	800	48.03	46.89	-1.18	$p \geq 0.25$

The AD test result presents a significant p-value, i.e., $p \geq 0.25$; it is a necessary condition but not a sufficient one for the case that the two distributions are the same. In other words, if a decision is made to reject the null hypothesis, that is, the distribution $F_M(x)$ is not close to the exponential distribution $G_N(x)$, such a decision will result in an error with a probability greater than 0.25.

D. APPLICATION OF AD TEST TO BGT DISTRIBUTION

In this subsection, we use the AD test to determine the distribution of the BGT of ETH-ECC. For this experiment, 90 threads were used to generate a block. We experimented using a fixed code length to observe the BGT without changing the difficulty level. In the test, two kinds of code length n are used: 32 and 36. These are the two lowest types of code length n in our pseudo-difficulty table used in the simulation. We divided the BGT into 10 intervals between the minimum BGT and maximum BGT for a histogram. For example, when the minimum BGT is 10 and the maximum BGT is 20, there are 10 intervals, i.e., [10,11], [11,12], ..., [19,20]. Using these intervals, we count the observed frequency of the BGT data. We set $F_M(x)$ using the observed frequency and set $G_N(x)$ using the mean of the BGT data. The mean in Fig. 4 is used for the expected frequency of $G_N(x)$ in Table 2. That is, the mean in Fig. 4 is used as $1/\lambda$ for the CDF of the exponential distribution $G_N(x)$:

$$G_N(x) = 1 - e^{-\lambda x} \tag{12}$$

The expected frequency of Table 1 is calculated using the integral of $G_N(x)$ corresponding to the interval time. Because $G_N(x)$ is the exponential distribution, if $F_M(x)$ is close to $G_N(x)$ we may consider $F_M(x)$ is an exponential distribution.

E. DISCUSSION ON AD TEST RESULTS

Fig. 4 shows the example result of the BGT over different blocks. Each block denotes the trial to obtain the BGT. We converted the test results, such as those in Fig. 4, to a distribution over time to analyze the BGT. These converted distributions are presented in Fig. 3. Fig. 3 presents the plots of the distribution of the observed and expected frequencies. These frequencies are calculated using the method described in Section V-D.

When we obtain a distribution using a sample set, there is a standard error; the standard error is high when the number of samples in the set is small. The standard error is expressed as

$$\frac{\sigma}{\sqrt{N}}$$

where σ denotes the standard deviation of a population and N denotes the cardinality of the sample set. The standard error decreases as the number of samples increases. Thus, the sample distribution becomes closer to the actual distribution of the observed samples. If the sample distribution, which reflects the actual distribution, differs from the expected distribution, we can observe that the sample distribution differs from the expected distribution. To observe the tendency of distribution over some blocks, we experimented with 100, 200, 300, and 400 blocks. Fig. 3 shows that the distribution of the observed frequency tends to follow the distribution of the expected frequency. In addition, Table 3 shows that the observed mean and standard deviation tend to converge as the number of blocks increases.

Furthermore, for the quantitative analysis, we use the AD test. Table 3 presents the AD test results to discuss hypotheses H_0 and H_A. These results show a similar result in Table 1 (c). In Table 1 (c), we drew samples from the same true distribution; the results present the largest possible p-value. All p-values in Table 3 are larger than or equal to 0.25, regardless of the number of blocks. In other words, if the null hypothesis is rejected, this decision will cause an error with a probability greater than 0.25. That is, the decision that the BGT distribution $F_M(x)$ does not follow the exponential distribution could be made with a high decision error.

VI. DISCUSSION

The purpose of ECCPoW is not to replace the current PoW of Ethereum. We propose our algorithm to present as one of the options for the Ethereum network. Ethereum can be utilized, for example, not only in a large-scale network but also in local-scale networks. To support a local-scale network, Ethereum provides PoW and PoA(Proof-of-Authority) as consensus algorithms. These algorithms have limitations for the local-scale network. For instance, PoW based local network has a risk of a double-spending attack by ASIC miners; PoA based network has a limitation of a participant because the time complexity of a PoA increases exponentially when the number of participants increases. Our algorithm, ECCPoW, can be utilized in such cases for the benefit of offering a novel PoW that allows numerous participants with deterrence to ASIC-borne attacks. In addition, our novel ECCPoW may open up for an expected use and thus untraveled future to Ethereum.

Extensive Simulation Set up at AWS. We have recruited twelve instances on Amazon Web Service (AWS) EC2; each instance of EC2 instances works as a node in a blockchain network. The cost of using AWS EC2 increases rapidly because PoW utilizes all the resources of instances. We were able to confirm that this scale of the experiment was good enough to achieve our main goal, which is aimed at verifying the stability of the block generation time of ECCPoW Ethereum. AWS simulation was done to obtain the trace data of block generation times. The twelve nodes employed in our simulation were divided into two different kinds of nodes. One kind is *bootnodes* which help the nodes connected. Nodes that want to join a network are connected to *bootnodes* first. After connection, *bootnodes* relay nodes to other nodes. In Ehtereum, *bootnodes* addresses are hardcoded on source codes, but it is possible to set *bootnodes* addresses manually for private networks. We have chosen two *bootnodes*. The other kind of nodes are *sealnodes* that participate in block generation as a miner in the PoW network. We have chosen the number of *sealnodes* to be 10. We use the m5.xlarge of AWS EC2, which has conventional node specification: four virtual CPUs and 16 GB memory for the real-world simulation. All nodes are deployed by Docker according to the guidance of Ethereum. Thus, all of our simulation results, which are shown in Fig. 2, are reproducible.

VII. CONCLUSION

In this work, we present the implementation, simulation, and validation of ETH-ECC. In the implementation, we showed how Ethereum can be updated with ECCPoW as its new consensus algorithm. In the simulation, we conducted a multinode experiment using AWS EC2. The results showed that ETH-ECC with its adaptive difficulty-level controllability is successfully implemented. In the validation, we showed statistical results in which the necessary condition for a finite mean BGT is satisfied such that the distribution of the ECCPoW block generation time is exponential.

REFERENCES

[1] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Accessed: Nov. 3, 2021. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] V. Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. Accessed: Nov. 3, 2021. [Online]. Available: https://ethereum.org/en/whitepaper/

[3] M. Rosenfeld, "Analysis of hashrate-based double spending," 2014, *arXiv:1402.2009*. [Online]. Available: http://arxiv.org/abs/1402.2009

[4] J. Jang and H.-N. Lee, "Profitable double-spending attacks," 2019, *arXiv:1903.01711*. [Online]. Available: http://arxiv.org/abs/1903.01711

[5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, Apr. 2014.

[6] S. Park, H. Choi, and H. Lee, "Time-variant proof-of-work using error correction codes," Jun. 2020, *arXiv:2006.12306*. [Online]. Available: https://arxiv.org/abs/2006.12306

[7] H. Jung and H. Lee, "Error-correction code based proof-of-work for ASIC resistance," *Symmetry*, vol. 12, no. 6, p. 988, Jun. 2020.

[8] W. Ryan and S. Lin, "Low-density parity-check codes," in *Channel Codes: Classical and Modern*. Cambridge, U.K.: Cambridge Univ. Press, 2009.

[9] R. G. Gallager, *Low Density Parity Check Cdes*. Cambridge, MA, USA: MIT Press, 1963.

[10] S. Shao, P. Hailes, T.-Y. Wang, J.-Y. Wu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "Survey of turbo, LDPC, and polar decoder ASIC implementations," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2309–2333, 3rd Quart., 2019.

[11] Y.-L. Ueng, B.-J. Yang, C.-J. Yang, H.-C. Lee, and J.-D. Yang, "An efficient multi-standard LDPC decoder design using hardware-friendly shuffled decoding," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 60, no. 3, pp. 743–756, Mar. 2013.

[12] E. Duffield and D. Diaz. *Dash: A Payments-Focused Cryptocurrency*. Accessed: Nov. 3, 2021. [Online]. Available: https://github.com/dashpay/dash/wiki/Whitepaper

[13] T. Black and J. Weight. *X16R ASIC Resistant by Design*. Accessed: Nov. 3, 2021. [Online]. Available: https://ravencoin.org/assets/documents/X16R-Whitepaper.pdf

[14] W. E. Ryan and S. Lin, *Channel Codes—Classical and Modern*. Cambridge, U.K.: Cambridge Univ. Press, 2009.

[15] S. Foss, D. Korshunov, and S. Zachary, *An Introduction to Heavy-Tailed and Subexponential Distributions*. New York, NY, USA: Springer, 2011.

[16] T. W. Anderson and D. A. Darling, "Asymptotic theory of certain 'goodness of fit' criteria based on stochastic processes," *Ann. Math. Statist.*, vol. 23, no. 2, pp. 193–212, Jun. 1952.

[17] A. N. Pettitt, "A two-sample anderson-darling rank statistic," *Biometrika*, vol. 63, no. 1, pp. 161–168, 1976.

[18] F. Scholz and M. Stephens, "K-sample anderson-darling tests," *J. Amer. Stat. Assoc.*, vol. 82, no. 399, pp. 918–924, Sep. 1987.

[19] W. G. Cochran, "The X^2 test of goodness of fit," *Ann. Math. Statist.*, vol. 23, no. 3, pp. 315–345, Sep. 1952.

[20] J. L. Hodges, "The significance probability of the Smirnov two-sample test," *Arkiv Matematik*, vol. 3, no. 5, pp. 469–486, Jan. 1958.

[21] W. G. Cochran, "Some methods for strengthening the common tests," *Biometrics*, vol. 10, pp. 417–451, Dec. 1954.

[22] J. J. Filliben. *1.3.5.16. Kolmogorov–Smirnov Goodness-of-Fit Test*. NIST/SEMATECH e-Handbook of Statistical Methods. Accessed: Nov. 3, 2021. [Online]. Available: http://www.itl.nist.gov/div898/handbook/

[23] T. Black. *Ravencoin–ASIC Thoughts—Round Two*. Accessed: Nov. 3, 2021. [Online]. Available: https://medium.com/@tronblack/ravencoin-asic-thoughts-round-two-f4f743942656

[24] Tevador. *Random X*. Accessed: Nov. 3, 2021. [Online]. Available: https://github.com/tevador/RandomX

[25] M. Belotti, N. Bozic, G. Pujolle, and S. Secci, "A vademecum on blockchain technologies: When, which, and how," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, pp. 3796–3838, 4th Quart., 2019.

[26] S. Asmussen, "Steady-state properties of of GI/G/1," in *Applied Probability and Queues*. New York, NY, USA: Springer, 2003, pp. 266–301, doi: 10.1007/0-387-21525-5_10.

HYOUNGSUNG KIM received the B.S. degree in electronics and computer engineering from Chonnam National University, Gwangju, South Korea, in 2019, and the M.S. degree from the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Gwangju. Since 2021, he has working with Korea Electronics Technology Institute. His research interests include blockchain, distributed systems, and statistical analysis.

JEHYUK JANG received the B.S. degree in electronic engineering from the Kumoh National Institute of Technology, Gumi, South Korea, in 2014, and the M.S. degree in information and communication engineering from GIST, South Korea, where he is currently pursuing the Ph.D. degree with the School of Electrical Engineering and Computer Science. His research interests include sub-Nyquist sampling and compressed sensing.

SANGJUN PARK received the B.S. degree in computer engineering from Chungnam National University, Daejeon, South Korea, in 2009, and the Ph.D. degree in electrical engineering and computer science from Gwangju Institute of Science and Technology, Gwangju, South Korea, in 2019. Since 2020, he has been working with the Electronics and Telecommunications Research Institute. His research interests include information theory, numerical optimization, compressed sensing, blockchain, deep-neural networks, and finite state machine.

HEUNG-NO LEE (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees from the University of California, Los Angeles, CA, USA, in 1993, 1994, and 1999, respectively, all in electrical engineering. From 1999 to 2002, he was a Research Staff Member with HRL Laboratories, LLC, Malibu, CA, USA. From 2002 to 2008, he was an Assistant Professor with the University of Pittsburgh, PA, USA. In 2009, he was with the School of Electrical Engineering and Computer Science, GIST, South Korea, where he is currently affiliated. His research interests include information theory, signal processing theory, blockchain communications/networking theory, and their application to wireless communications and networking, compressive sensing, future internet, and brain–computer interface. He was a recipient of several prestigious national awards, including the Top 100 National Research and Development Award, in 2012; the Top 50 Achievements of Fundamental Researches Award, in 2013; and the Science/Engineer of the Month, in January 2014.