

Thesis for Master's Degree

# Error Correction Code Based Proof-of-Work on Ethereum

Hyongsung Kim

School of Electrical Engineering and Computer Science

Gwangju Institute of Science and Technology

2021

석사학위논문

# 이더리움에 대한 오류 정정 부호 기반 작업 증명

김형성

전기전자컴퓨터공학부

광주과학기술원

2021

# Error Correction Code Based Proof-of-Work on Ethereum

Advisor: Professor Heung-No Lee

by

Hyongsung Kim

School of Electrical Engineering and Computer Science


Gwangju Institute of Science and Technology

A thesis submitted to the faculty of the Gwangju Institute of Science and Technology in partial fulfillment of the requirements for the degree of Master of Science in the School of Electrical Engineering and Computer Science

Gwangju, Republic of Korea

2020. 12. 07

Approved by



Professor Heung-No Lee

Committee Chair

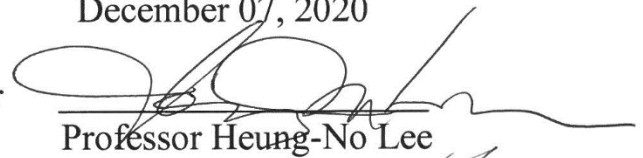
# Error Correction Code Based Proof-of-Work on Ethereum

Hyoungsung Kim

Accepted in partial fulfillment of the requirements for  
the degree of Master of Science

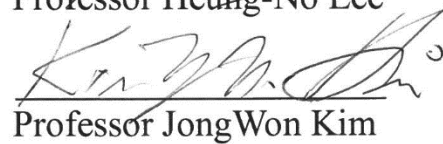
December 07, 2020

Committee Chair



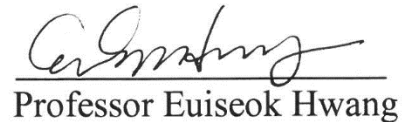
Professor Heung-No Lee

Committee Member



Professor JongWon Kim

Committee Member



Professor Euseok Hwang

## Abstract

These days, because of the centralization problem of proof-of-work (PoW), many researchers propose application-specific integrated circuit (ASIC)-resistant PoW and alternative consensus algorithm (e.g., proof-of-stake, delegated proof-of-stake, and Byzantium fault tolerance). However, networks of these alternative consensus algorithms present less decentralization than ASIC-resistant PoW. Specifically, in alternative consensus algorithms, a limited participant can generate blocks; but in ASIC-resistant PoW, anyone can join to generate a block. Thus, ASIC-resistant PoW presents a better-decentralized network than alternative algorithms. In this work, we utilize error-correction code based proof-of-work (ECCPoW) as known as ASIC-resistant PoW. The ECCPoW utilizes a low-density parity-check (LDPC) code that has flexible parameters: variable length code, continuously changed parity check matrix (PCM). Thus, ECCPoW is possible to impair ASIC by changing the parameter of LDPC. Previous researches on ECCPoW algorithms present its theory and the implementation on Bitcoin. However, they do not discuss the stability of its block generation time. Block generation time (BGT) must follow a distribution that has a finite mean to achieve the stability that can ensure confirmation of transactions. In ECCPoW algorithms, BGT follows a long-tailed distribution due to varying cryptographic puzzles. If this long-tailed distribution has a none finite mean, such as the heavy-tailed distribution, the confirmation of transactions is not guaranteed. Thus, validating the distribution of BGT is necessary to determine if consensus algorithms can guarantee the confirmation of transactions. In this work, we present the implementation, simulation, and stability validation of ECCPoW Ethereum. In the implementation, we demonstrate how Ethereum applies ECCPoW algorithm as a consensus algorithm. Moreover, in simulation, we perform a multinode simulation to show how ECCPoW algorithm works on Ethereum with difficulty change. In stability validation, to present moderate evidence if the BGT

has a finite mean, we obtain a goodness-of-fit result using the Anderson-Darling test. Our implementation is at GitHub<sup>1</sup>.

---

<sup>1</sup> <https://github.com/cryptoecc/ETH-ECC>

© 2020

Hyongsung Kim

**ALL RIGHTS RESERVED**

# Contents

<b>Abstract .....</b>	<b>v</b>
<b>Contents.....</b>	<b>viii</b>
<b>List of Figures .....</b>	<b>x</b>
<b>List of Tables.....</b>	<b>xi</b>
<b>1 Introduction.....</b>	<b>1</b>
1.1 Research background .....	1
<b>2 Consensus Algorithms and Its Comparison .....</b>	<b>5</b>
2.1 Proof-of-Work (PoW) .....	5
2.2 Proof-of-Stake (PoS).....	6
2.3 Practical Byzantium Fault Tolerance (PBFT).....	6
2.4 Comparison .....	7
<b>3 Requirement of ASIC-Resistance in PoW .....</b>	<b>10</b>
3.1 Intentional Bottleneck .....	10
3.2 High Complexity of ASIC Design .....	10
3.3 Combine Two Ways.....	11
<b>4 ECCPoW Implemented on Ethereum.....</b>	<b>13</b>
4.1 Overview of ECCPoW .....	13
4.2 ECCPoW on Ethereum .....	13
4.2.1 Variable Length Code.....	17
4.2.2 Continuously Changed Parity Check Matrix .....	17
4.2.3 Proof-of-Work of the LDPC decoder .....	18
4.2.4 Difficulty Control of ECCPoW Ethereum.....	20
<b>5 Problem Formulation .....</b>	<b>24</b>
<b>6 Experiment On ECCPoW Ethereum .....</b>	<b>25</b>



6.1	Experiment Set Up .....	25
6.1.1	Experiment set up on Linux .....	25
6.1.2	Experiment set up on Windows .....	34
6.1.3	How to connect nodes.....	36
	38	
6.2	Simulation of The Difficulty Change.....	38
6.3	Stability of The Block Generation Time .....	39
6.3.1	Anderson-Darling test.....	40
6.3.2	Experimental detail .....	45
6.3.3	Experimental result .....	46
<b>7</b>	<b>Conclusion .....</b>	<b>48</b>
	<b>References.....</b>	<b>49</b>
	<b>Acknowledgement.....</b>	<b>52</b>

## List of Figures

Figure 1 Structure of Blockchain.....	2
Figure 2 Trilemma and Consensus algorithms.....	3
Figure 3 Structure of Proof-of-Work.....	4
Figure 4 Message complexity comparison of PBFT and PoW.....	6
Figure 5 Overview of consensus algorithms.....	7
Figure 6 Example of off-chain solution .....	8
Figure 7 Scheme of ECCPoW Ethereum.....	11
Figure 8 Difficulty control mechanism of Ethereum .....	14
Figure 9 The heaviest chain rule of Ethereum .....	15
Figure 10 The Longest chain rule of Bitcoin.....	15
Figure 11 Difficulty control mechanism of Bitcoin.....	16
Figure 12 Example of parity check matrix.....	18
Figure 13 Simulation of ECCPoW .....	20
Figure 14 Example case of rollback.....	21
Figure 15 Problem of the heavy-tailed distribution .....	23
Figure 16 Binary files in bin folder.....	26
Figure 17 Result of storage directory initialization.....	29
Figure 18 VMware .....	35
Figure 19 Step to install Ubuntu .....	35
Figure 20 Update package .....	36
Figure 21 Install golang.....	36
Figure 22 Install git.....	36
Figure 23 Connected peer list .....	38
Figure 24 Histogram of observed frequency and expected frequency .....	39
Figure 25 Block generation time of 32 code length for 300 blocks .....	45

## List of Tables

Table 1 Example of the Anderson-Darling test results .....	42
Table 2 The observed frequency and expected frequency over time .....	44
Table 3 Anderson-Darling test results.....	46

# Chapter 1

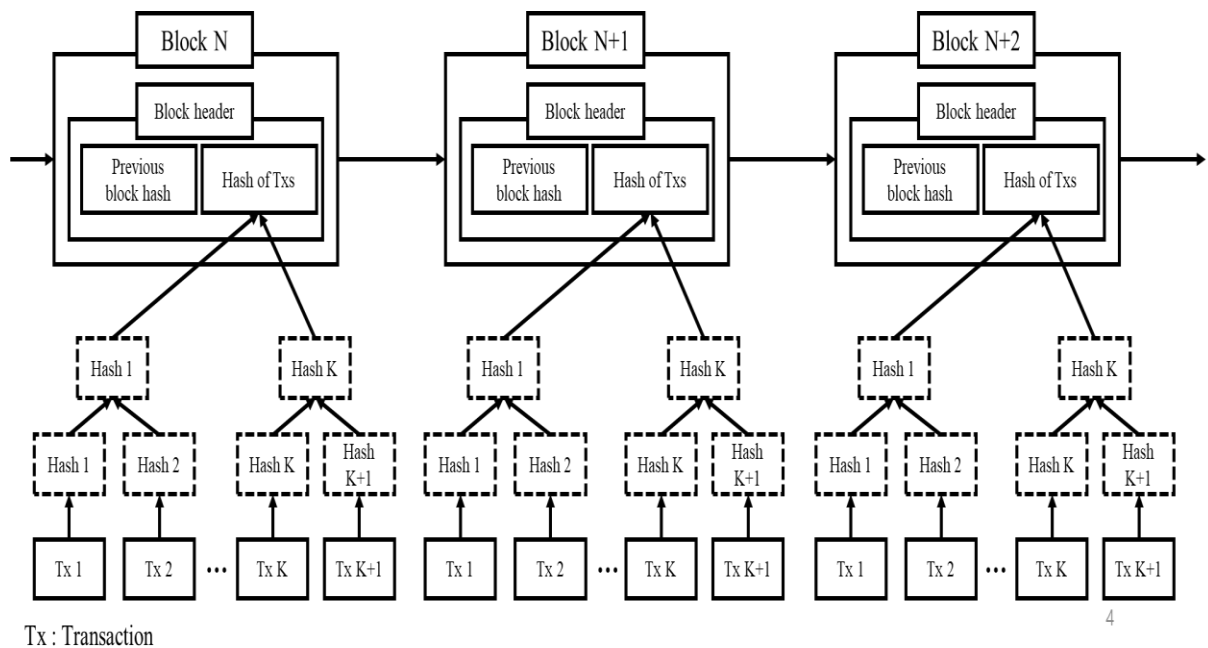
## 1 Introduction

### 1.1 Research background

Unlike reliable network which consists of trust nodes, decentralized network consists of trustless nodes. In reliable network, there is no peers that would send wrong information on purpose. However, in decentralized network, it must be considered for possibility that there are some peers who may send wrong or forged information to others. To consider these issues in decentralized network, Nakamoto has presented blockchain and consensus algorithm [1].

In blockchain of Nakamoto, there are miner nodes who generate a block which is including transactions. When one of miner generate a block, block is propagated to other miners with hash of transactions and miners who received a block validate it. If validation is done, block is linked to previous block like chain. Figure1 shows structure of blockchain. Nakamoto considered that blockchain is utilized by trustless nodes so one of node may send false or forged information to other peers. Thus, Nakamoto proposed proof-of-work (PoW) as a consensus algorithm. In PoW, there exists crypto puzzle. All of miners try to solve a puzzle by using hash function (e.g., SHA256). When a miner finds a hash value from hash function which can meet a condition of crypto puzzle, then miner sends a block with hash value to other miners. Other miners validate that the sender really solves a crypto puzzle. If it solves, then valid block is generated and linked to previous block. The generated block includes information of previous block. Thus, if someone want to change a one of block in chain, all previous block of changed block also must be changed. Therefore, it is impossible to send a block which is including false or forged information to other nodes unless the network is centralized by a particular group.

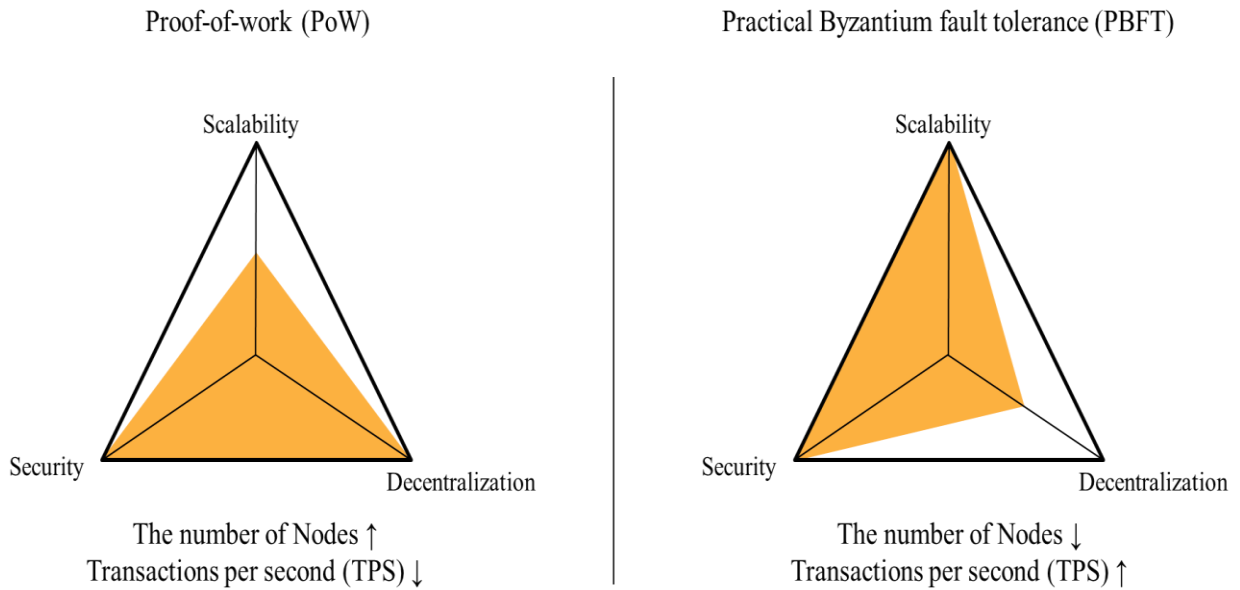
PoW needs a lot of computational power. Thus, when a miner success to generate a block, miner get a kind of incentive to reimburse a cost which is derived by computational work. In blockchain of Nakamoto, incentive is called as Bitcoin. In the early days of Bitcoin, the price of incentive is not high. However, as the price of the incentive increased, miners who want to monopolize incentives have appeared. They use application specific integrated circuit (ASIC) which has higher computational power



**Figure 1 Structure of Blockchain**

than general purpose unit such as central processing unit (CPU) or graphic processing unit (GPU). As a result, network of PoW blockchain is centralized by ASIC miners. In centralized network, a minority of nodes which occupy a most computational power monopolize the chance to generate blocks. Centralized network yields vulnerability to blockchain network. Because consensus algorithm cannot suppress malicious attack such as filtering out some transactions or 51% attack in centralized network [2], [3].

Since Bitcoin experienced centralization problem, Buterin and Wood proposed ASIC-resistant PoW which is called as Ethash with Ethereum [4], [5]. In Ethash, miners must mix a data which are stored in memory. Thus, there is a bottleneck between arithmetic logic unit (ALU) and memory. Even though ALU such as ASIC has strong computational power, the number of attempts to generate a block depends on bandwidth of memory. It has shown ASIC-resistant for a long time. However, in 2018, Bitmain released ASIC for Ethash. Not only Ethash but also there are various consensus algorithms which are designed for ASIC-resistant. All of them have at least one of these features. One is *intentional bottleneck* such as Ethash and the other is *high complexity of design* such as X11 of Dash [6] or X16R Raven [7].



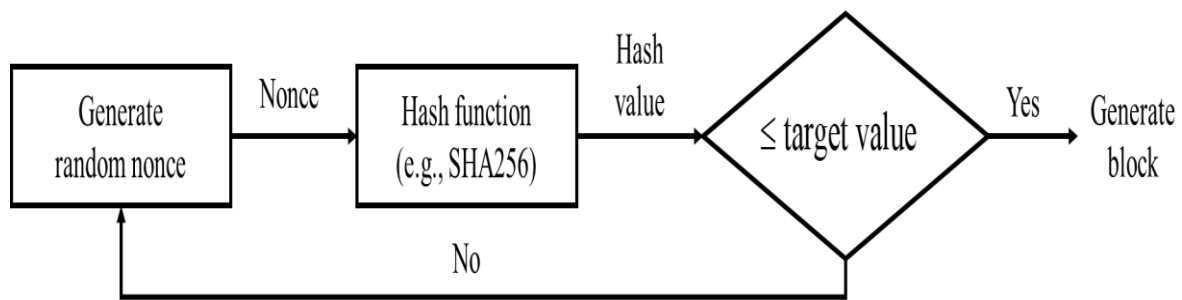
**Figure 2 Trilemma and Consensus algorithms**

In Bitcoin, there exist only one hash function. However, in X11 and X16R use more hash functions to make design ASIC hard. More details are described in Chapter 2.

To utilize error-correction code (ECC) for ASIC-resistance, in [8], they proposed how to apply ECC to PoW and in [9], they showed that ECC based PoW (ECCPoW) can be applied to real world by presenting implementation on Bitcoin. In [8], they defined that block generation follows geometric distribution with the number of try to generate a block. However, in [9], they mentioned that ECCPoW has an unstable block generation time. Namely, block generation time follows long-tailed distribution. If this long-tailed distribution follows a distribution which has none finite mean such as a heavy-tailed distribution [10], it cannot be expected that when block will be generated. As a result, geometric distribution of block generation which is defined in [8] is not guaranteed. In this paper, we present a way to apply ECCPoW to Ethereum. Also, we discuss stability of distribution of block generation time of ECCPoW which is not presented in [8] and [9].

The contributions of my work are listed as follows:

- We present how to implement ECCPoW on Ethereum



**Figure 3 Structure of Proof-of-Work**

- We present how to control difficulty of block generation in ECCPoW Ethereum and show simulation of ECCPoW Ethereum with difficulty change
- We present moderate evident that exponential distribution describes distribution of block generation time of ECCPoW Ethereum by showing goodness-of-fit result which is derived by Anderson-Darling test

The remainder of this paper is organized as follow: in Chapter 2, I present brief introduction of consensus algorithms and comparison with PoW. In Chapter 3, I present requirements of ASIC-resistant PoW with example. In Chapter 4, we explain implementation of ECCPoW on Ethereum. In Chapter 5, we formulate a problem. In Chapter 6, we present simulation results and statistical analysis. In Chapter 7, We summarize my work and present conclusion.

# Chapter 2

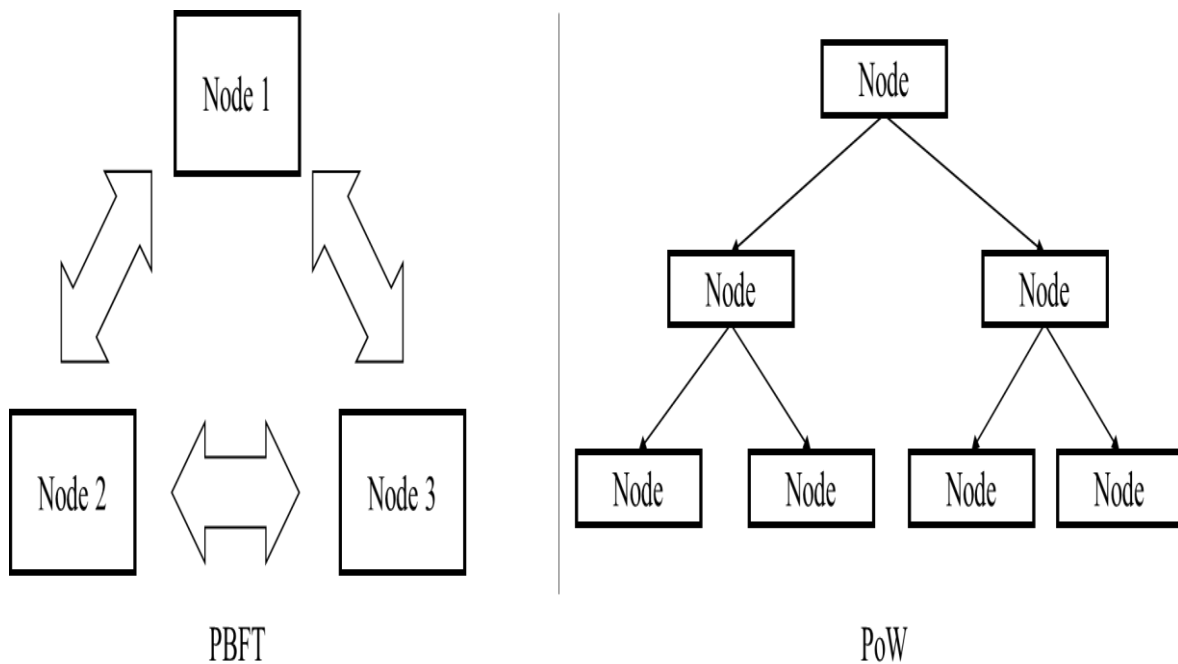
## 2 Consensus Algorithms and Its Comparison

In this Chapter, we briefly review commonly used consensus algorithm and present why we must use proof-of-work (PoW) for decentralized network. In blockchain, there is a trilemma that security, scalability and decentralization cannot be achieved at the same time. Thus, consensus algorithm gives up one of things in trilemma. PoW gives up scalability but it achieves decentralization and security. Proof-of-Stake (PoS) cannot achieve scalability too. Practical Byzantium fault tolerance achieves decentralization, but it gives up decentralization. Figure 2 shows trilemma according to consensus algorithms. More detail of these consensus algorithms and comparison are explained in following subchapter.

### 2.1 Proof-of-Work (PoW)

PoW is proposed by Nakamoto for Bitcoin [1]. In PoW, there exists a crypto puzzle and miners. Crypto puzzle is kind of puzzle which can be solved by brute force; miners are defined as nodes who try to solve crypto puzzle. To solve a puzzle, miner use hash function. After getting an output of hash function, miner compare output with target value of crypto puzzle. When miner success to find a hash value which is lower than target value, then miner can publish a block. Figure 3 presents structure of PoW. One of benefits of PoW is that PoW has  $O(1)$  message complexity [11]. Namely, even though there are a lot of nodes, sending only 1 message is enough for PoW network. Thus, PoW can increase the number of nodes for network. However, in PoW, block is generated by competition. When blocks are generated at the same time, it is called as fork. If blocks are generated at same time, only one block will be selected and transactions which are included in non-selected blocks are rolled back. Because of roll back, PoW cannot generate a block fast. For example, if block is generated every 1 second, then forks and roll backs will be happened frequently. Thus, even though PoW has good decentralization, there is a limit to increase transactions per second (TPS). Because of low TPS, PoW cannot meet scalability of trilemma.





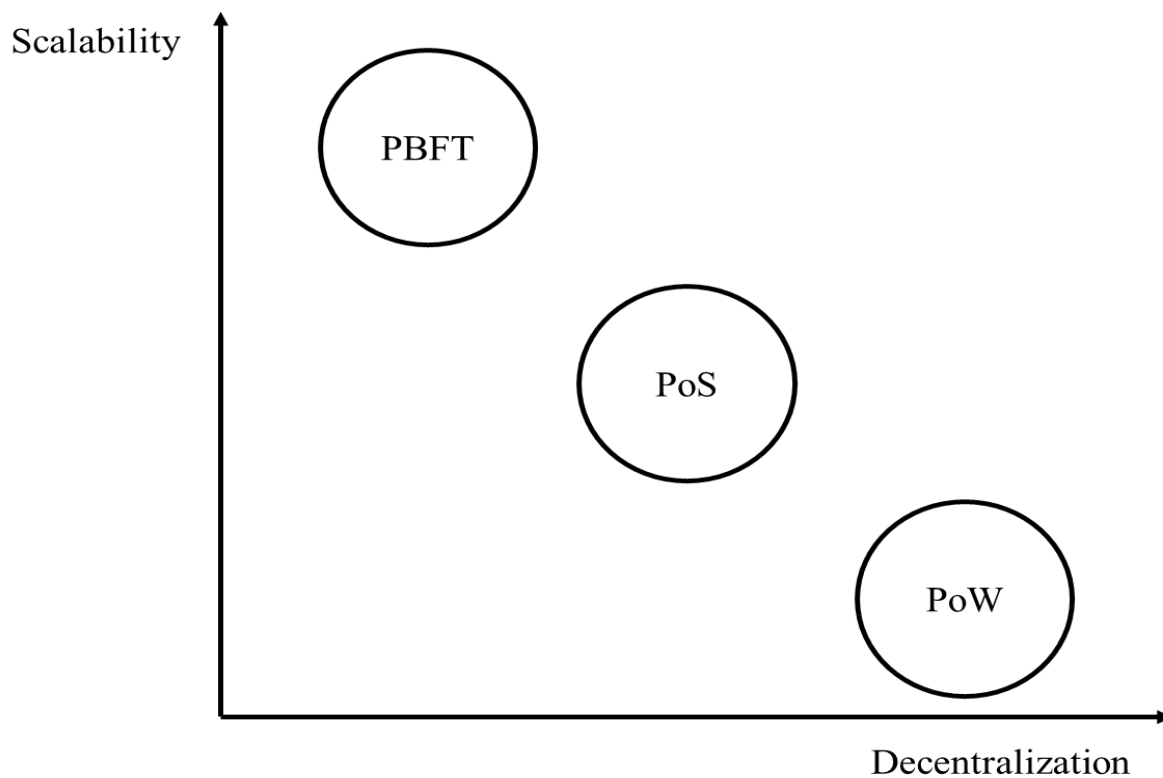
**Figure 4 Message complexity comparison of PBFT and PoW**

## 2.2 Proof-of-Stake (PoS)

In PoS, block generation depend on quantity of stake. The time which takes to generate a block is called round. There is only one miner who generate a block in one round and this miner is called as proposer or leader. Proposer is changed every round depend on quantity of stake. When there are 100 rounds and one miner has 30 % of total network staking, then probabilistically, this miner generates about 30 blocks. Unlike PoW, there is block proposer to generate a block. Thus, PoS has better scalability than PoW. Also, PoS has  $O(1)$  message complexity like PoW [11]. However, it has less decentralization than PoW. Because in PoW, anyone can join to block generation, but in PoS, there is a minimum stake for generating a block in PoS.

## 2.3 Practical Byzantium Fault Tolerance (PBFT)

PBFT [14] is a consensus algorithm which is based on Byzantium Fault Tolerance (BFT) [12], [13]. BFT is proposed to consensus when malicious nodes exist and PBFT defines that it needs  $3N + 1$

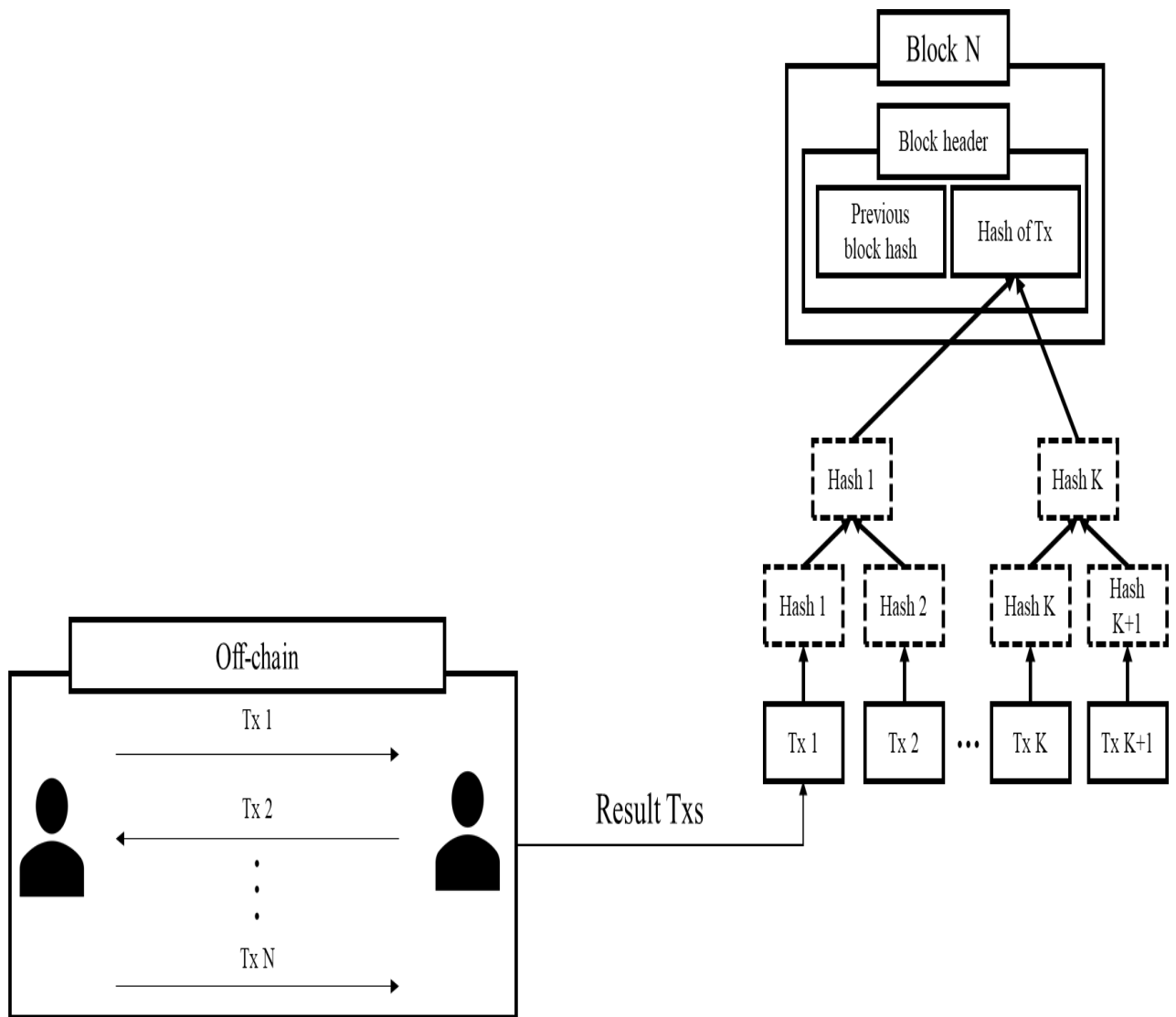


**Figure 5 Overview of consensus algorithms**

nodes when there exists  $N$  malicious nodes. In PBFT, there are three steps: pre-prepare, prepare, commit. Thus, it has  $O(n^2)$  of message complexity. It means when there are 10 nodes, PBFT has 100 times message exchange. Thus, there is a limit to the number of nodes so it has lower decentralization than PoW. However, there is no fork so PBFT has higher scalability than PoW. Figure 4 presents different of message complexity between PBFT and PoW.

## 2.4 Comparison

We briefly introduced consensus algorithms which are commonly used in blockchain. In this sub-chapter, we present comparison of aforementioned consensus algorithm and explain why PoW is important. We have mentioned that PoW has more decentralization than any other consensus algorithms. Because PoW can increase the number of nodes which can generate a block. However, PBFT has limitation of the number of nodes. Because PBFT has  $O(n^2)$  as message complexity. Thus, when the number of nodes increase, overhead of network also increase. PoS is decentralized than PBFT and anyone can



**Figure 6 Example of off-chain solution**

join like PoW. However, there is a minimum stake to join mining. Thus, for decentralized blockchain network, PoW is utilized as consensus algorithm. However, PoW has lower scalability. Namely, PoW has low transaction per second (TPS). Thus, PoW needs a solution to increase TPS in future work such as off-chain solution. Figure 6 shows example of off-chain solution. It reports only last transaction to main blockchain network.

Even though PoW has good decentralization, there are many blockchains which are not using PoW as consensus algorithm due to appearance of ASIC. ASIC has better efficiency than general purpose unit (e.g., CPU, GPU). As a result, miners who utilize ASIC dominate mining of PoW and network is

centralized by ASIC nodes. To solve centralization problem, ASIC-resistant PoW is needed.

# Chapter 3

## 3 Requirement of ASIC-Resistance in PoW

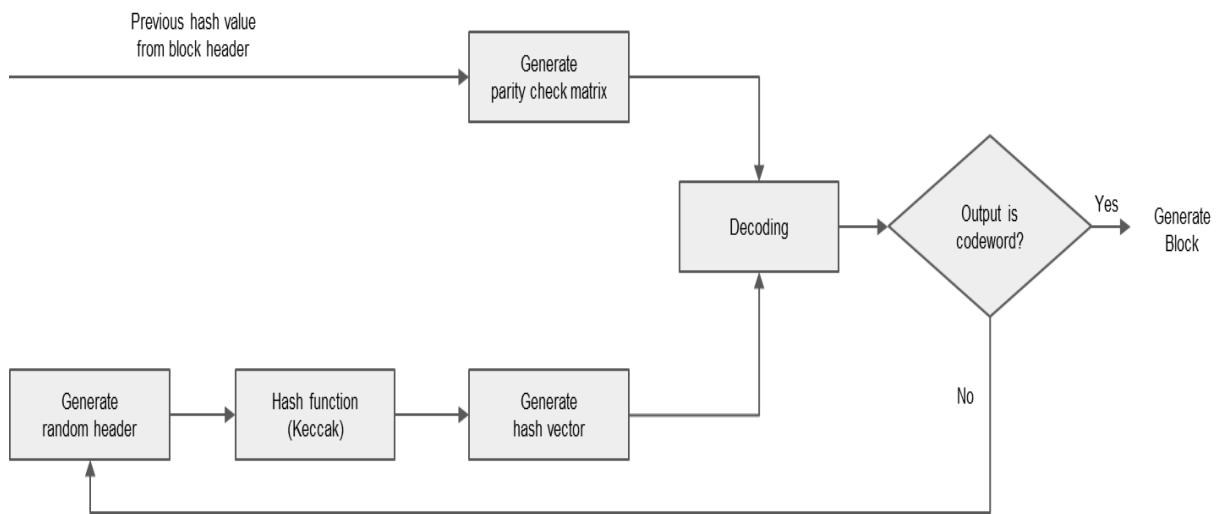
In this Chapter, we introduce requirements of ASIC-resistant in PoW with example. ASIC-resistant PoW utilizes at least one of these two methods: *intentional bottleneck* and *high complexity of ASIC design*. The *intentional bottleneck* which is also termed as memory-hard technique. The *Intentional bottleneck* uses bottleneck between arithmetic logic unit (ALU) and memory. This method is used in Ethash of Ethereum [4], [5]. The *High complexity of ASIC design* is make ASIC design hard; for example, it uses sequence of hash functions unlike Bitcoin which uses only 1 hash function. This method is used in X11 of Dash [15] or X16R of Raven [16]. Furthermore, Random X of Monero [17] combines both methods for ASIC-resistant. More detail of these methods are explained in next Chapters.

### 3.1 Intentional Bottleneck

This method uses bottleneck between arithmetic logic unit (ALU) and memory. Even though ALU has high throughput like ASIC, if memory has not enough bandwidth, ALU cannot fully utilize throughput of ALU. Specifically, if miner has to use a data which are stored in memory, the number of attempts to generate a block depend on bandwidth of memory. The most known PoW of *intentional bottleneck* is Ethash of Ethereum. Ethash uses a directed acyclic graph (DAG) which cannot store in cache memory; therefore, DAG is stored in memory. Miners who want to generate a block in Ethash have to mix a data of DAG. This method has been ASIC-resistant for a long time. However, ASIC is released in 2018 by Bitmain.

### 3.2 High Complexity of ASIC Design

This method makes ASIC design hard such as using sequence of hash function. Namely, purpose of this method is to be less efficient compared with the general purpose unit (e.g., CPU and GPU). For example, if it is expensive to design an ASIC with the same performance as that of a CPU or GPU than buying CPU or GPU, there is no reason to design and develop an ASIC. X11 or Dash [15] and X16R of Raven [16] are utilizing this method. X11 uses 11 hash functions: BLAKE, BMW, Grostel, JH,



**Figure 7 Scheme of ECCPoW Ethereum**

Keccak, Skein, Luffa, Cubehash, SHAvite-3, SIMD and ECHO. First hash function; BLAKE uses header of blocks as an input and pass an output to next hash function. Using the result of ECHO, miner determines whether the miner obtains a correct nonce. X11 had been shown to be ASIC resistance but Bitmain released an ASIC for X11 in 2016. There are few PoW algorithms which extend X11 (e.g., X13, X14 and X15); however, the ASIC for these have been released. X16R is also extended PoW of X11. Unlike X11, it randomly shuffles 16 hash functions. However, T. Black, who designed X16R, mentioned that it is evident that there are ASICs for X16R [18]. Our method ECCPoW gets ASIC-resistance by using high complexity of ASIC design. However, unlike previous research, ECCPoW uses continuously changed crypto puzzles. We give more detail in next Chapter.

### **3.3 Combine Two Ways**

Random X of Monero combines two aforementioned methods. Random X uses memory-hard techniques for bottleneck with random code execution [17]. Random code execution is optimized for CPU. Even though it is possible to perform mining using FPGA or ASIC but it is less efficient than CPU mining. In other word, currently mining random X using FPGA or ASIC is inefficient. However, in future, if chip price decrease, FPGA or ASIC mining can surpass CPU mining. Also ASIC of ECCPoW

can be released when chip price decreased. However, by changing parameters which are used for mining, it is possible to make ASIC powerless.

# Chapter 4

## 4 ECCPoW Implemented on Ethereum

In this Chapter, we introduce ECCPoW and present how it is implemented on Ethereum with Figure 7. Also, we present how difficulty of ECCPoW Ethereum is controlled.

### 4.1 Overview of ECCPoW

Blockchains which are utilizing PoW as consensus algorithm, nodes must solve a crypto puzzle to generate a block. The first node who solve a puzzle first get an authority to generate a block. This process to solve a puzzle is called as mining. In Bitcoin, crypto puzzle is finding a specific output of Secure Hash Algorithms (SHA) which is lower than target hash value. Ethereum uses a similar puzzle to that of the Bitcoin by replacing SHA with Ethash.

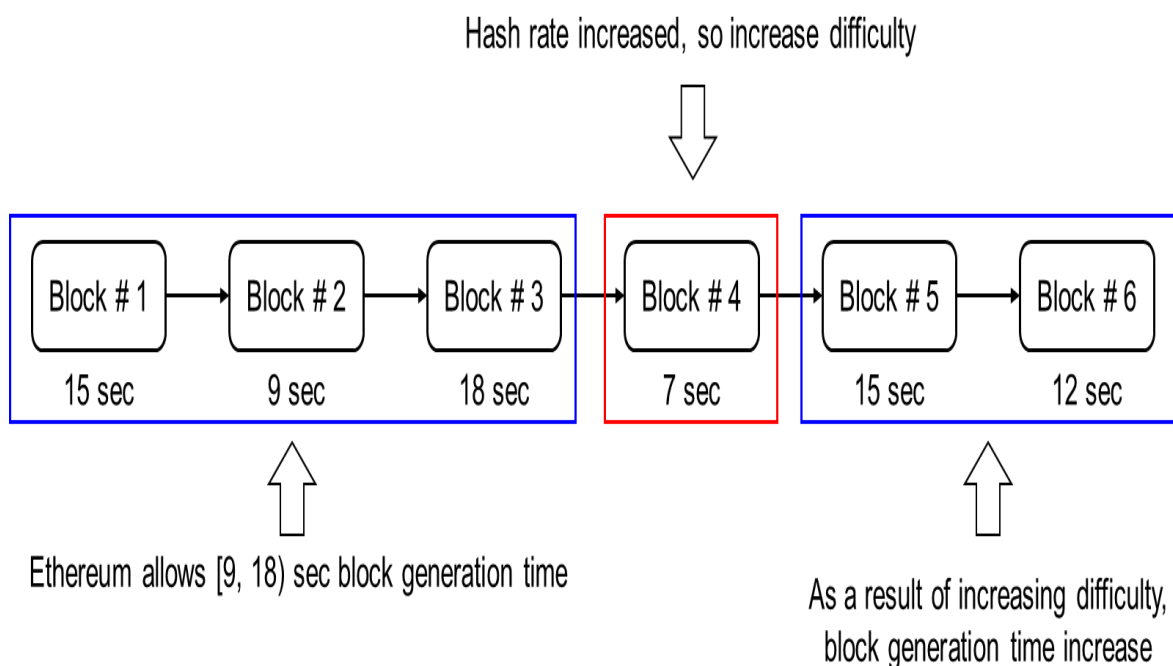
The ECCPoW algorithm proposed in [8] utilizes error-correction code for PoW. There are two factors that make ECCPoW ASIC-resistance: variable length code and random puzzle generator. ECCPoW can generate variable length code by slicing or concatenating result of hash function. Furthermore, unlike previous research for ASIC-resistant PoW, ECCPoW generate continuously changed crypto puzzle by utilizing random puzzle generator. A random puzzle generator is defined by parity check matrix (PCM). In ECCPoW, nodes generate a new PCM which is distinct from other previously generated PCM. Also, PCM is generated by previous block. Thus, it is impossible to generate a PCM in advance. Figure 7 shows above steps; more details are discussed in the next subchapter.

### 4.2 ECCPoW on Ethereum

In this subchapter, we present how error-correction code is applied to proof-of-work with Figure 7. Our definitions of this Chapter are based on [8]. We utilize low density parity check (LDPC) code for error correction.

$$C := \{ \mathbf{c} \mid \mathbf{H}\mathbf{c} = \mathbf{0} \cap \mathbf{c} \in \{0,1\}^{n \times 1} \} \quad (1)$$





**Figure 8 Difficulty control mechanism of Ethereum**

When a parity check matrix (PCM)  $\mathbf{H}$  is given, a vector  $\mathbf{c}$  which satisfies (1) is referred to as an LDPC code. The goal of ECCPoW is succeeding a LDPC decoding when a PCM and hash vector are given to the LDPC decoder. These steps are described in Figure 7. We use variable length code and continuously changed PCM for ASIC-resistance. A Vector  $\mathbf{c}$  which is used as a code has variable length code. We denote how to generate variable length code in next subchapter. A PCM is randomly generated but all of miners use same PCM until the block is generated. Specifically, we randomly generate the PCM using the Gallagher's way [19] and a previous hash value. A previous hash value is used as a seed when matrix is shuffled in Gallagher way. More details are presented in 4.2.2.

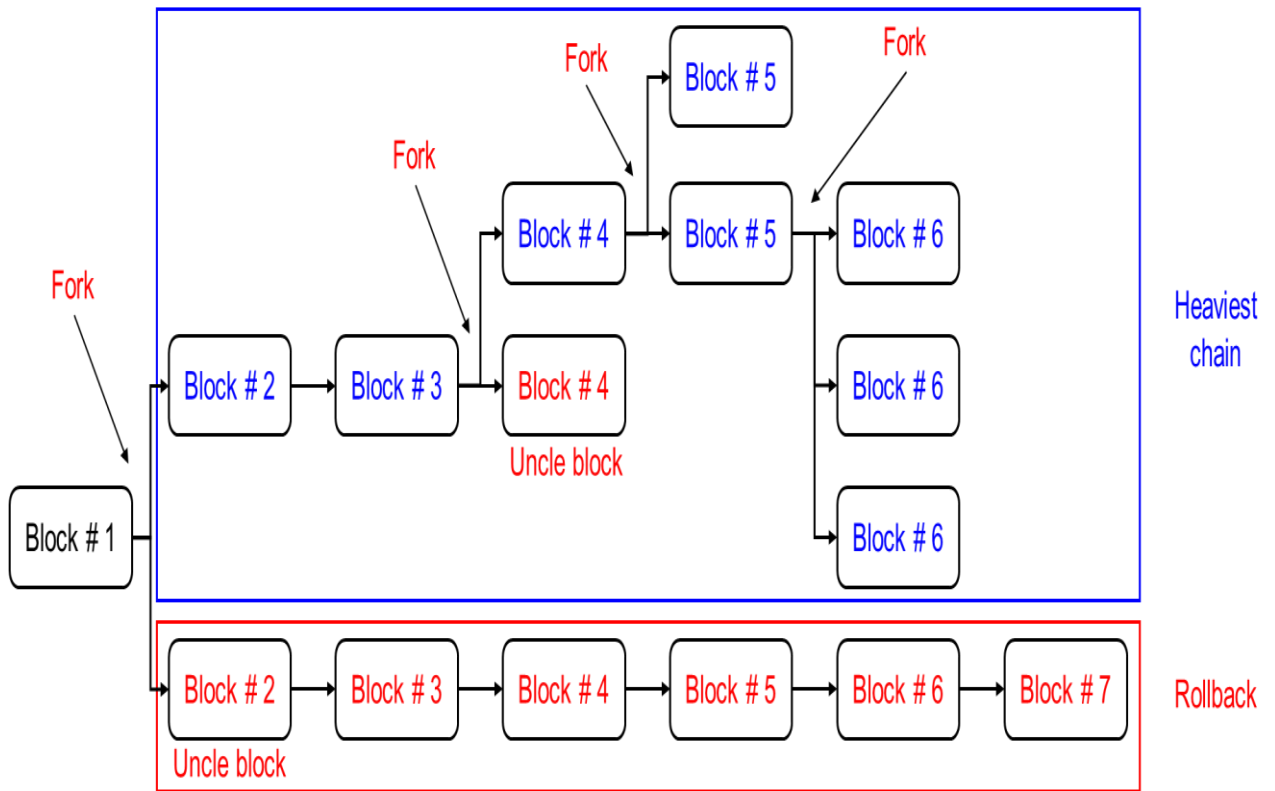


Figure 9 The heaviest chain rule of Ethereum

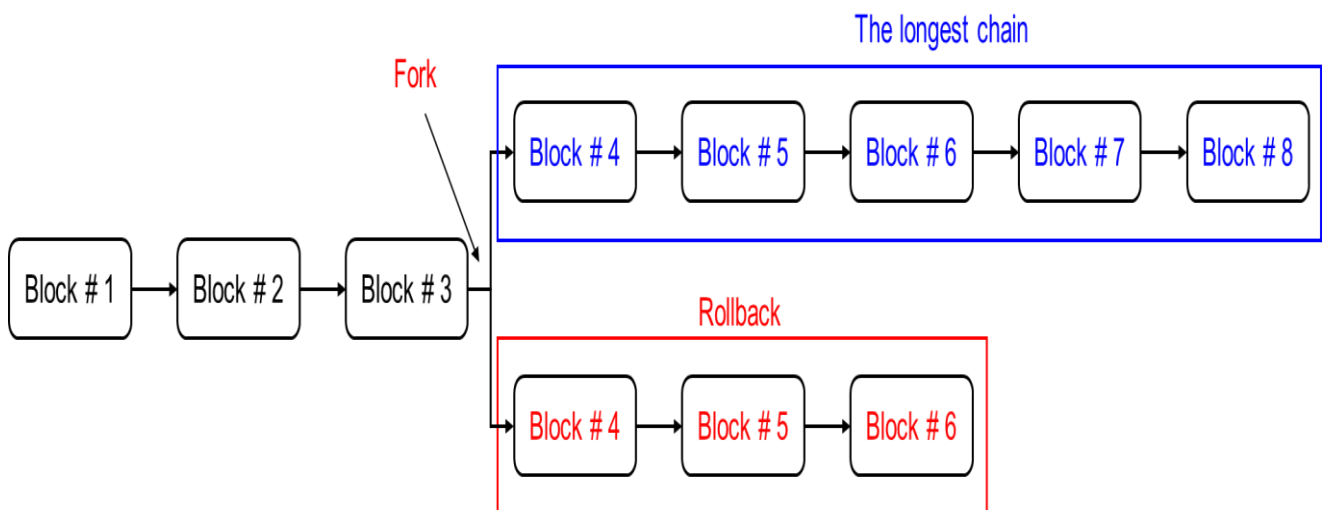
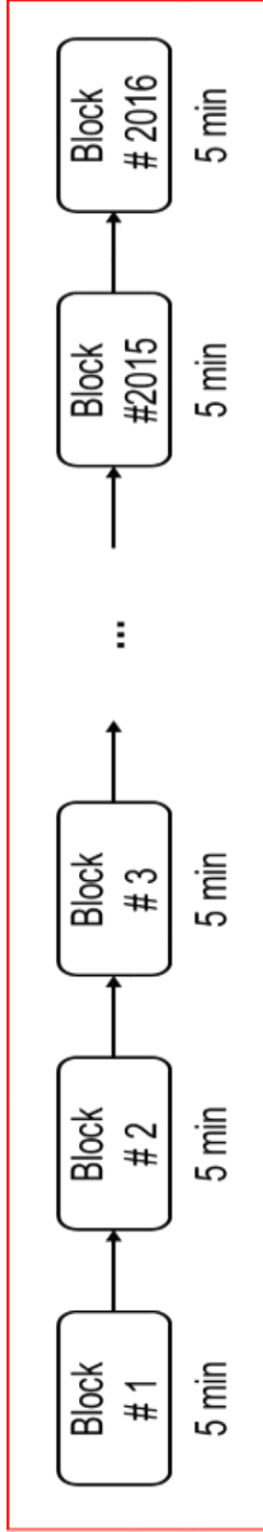
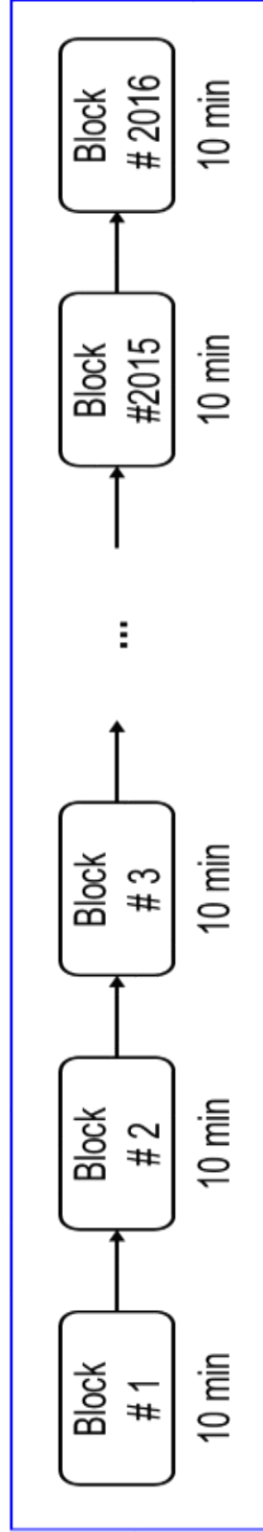


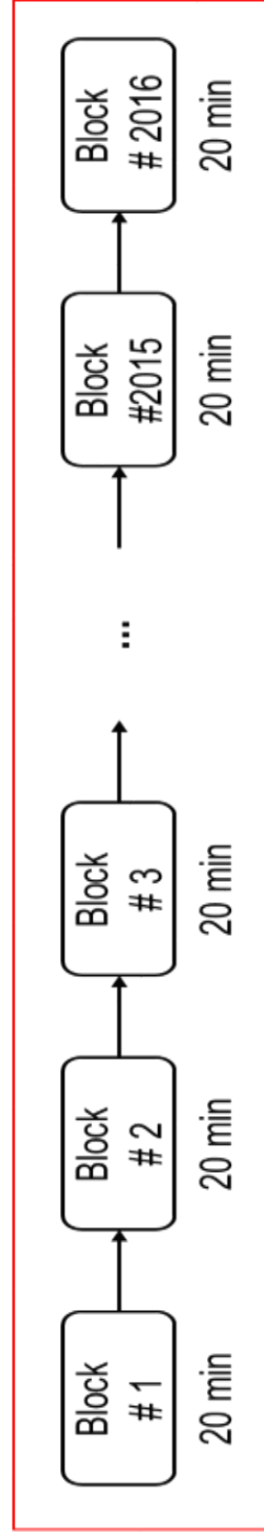
Figure 10 The Longest chain rule of Bitcoin



(a) 1 week for 2016 blocks



(b) 2 weeks for 2016 blocks



(c) 4 weeks for 2016 blocks

Figure 11 Difficulty control mechanism of Bitcoin

### 4.2.1 Variable Length Code

In this subchapter we present how to get a variable length code. This subchapter corresponds to Generate hash vector of Figure 7.

**Definition 1.** Hash vector  $\mathbf{r}$  in which the size of  $n$  can be obtained as follows:

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

Where *Keccak* denotes the hash function applied in Ethash of Ethereum, and nonce is randomly generated in the same way as Ethash.

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

Where  $l = \lfloor n / 256 \rfloor$  and  $j = n - 256 \times l$ . For example, when  $n$  is less than 256,  $\mathbf{r}$  gets the same length as  $n$  by slicing  $s_1$ , and when  $n$  is not less than 256, the results are concatenated with  $s_l := \text{Keccak}(s_{n-1})$ .

### 4.2.2 Continuously Changed Parity Check Matrix

For PCM, there are few conditions which must meet. First, all of nodes must use same PCM without broadcasting. Second, PCM has to be changed every block. For these condition, we use Gallagher's way [19] with previous hash vector. Below steps show example of how to generate PCM which has: two ones in column, four ones in row and 12 length of code

**Step 1:** Construct a matrix such as

$$\mathbf{A}_1 := \begin{bmatrix} \mathbf{1}_4 & \mathbf{0}_4 & \mathbf{0}_4 \\ \mathbf{0}_4 & \mathbf{1}_4 & \mathbf{0}_4 \\ \mathbf{0}_4 & \mathbf{0}_4 & \mathbf{1}_4 \end{bmatrix}$$

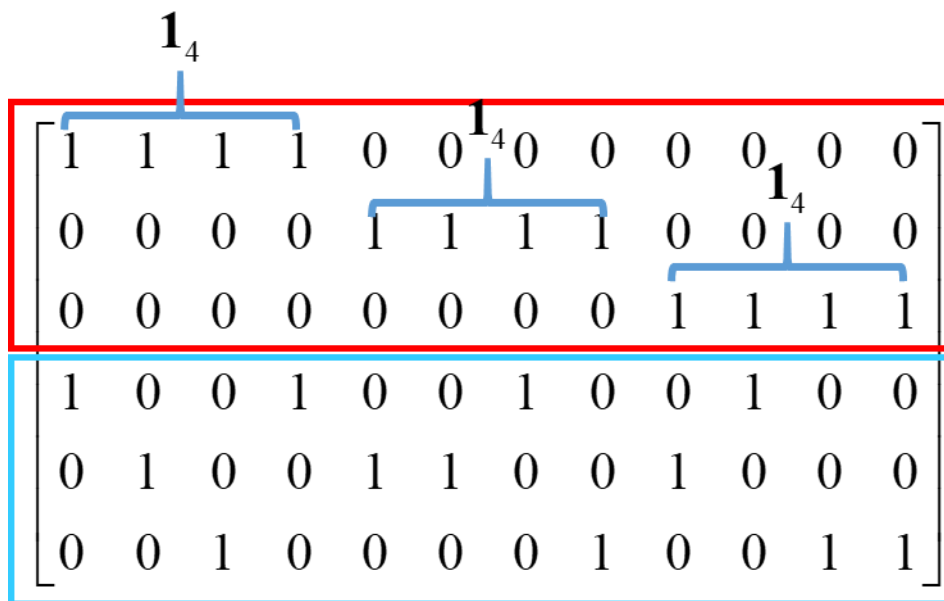
Where  $\mathbf{1}_4 := [1 \ 1 \ 1 \ 1]$  and  $\mathbf{0}_4 := [0 \ 0 \ 0 \ 0]$ .

**Step 2:** Permute  $\mathbf{A}_1$  randomly to form  $\mathbf{A}_2$

For permutation, we use previous hash vector as a seed. Thus,  $\mathbf{A}_2$  is randomly generated but all nodes can generate same PCM.

**Step 3:** Construct a PCM

$$\mathbf{H} := \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix}$$



**Figure 12 Example of parity check matrix**

The Figure 11 shows result of above steps.

#### 4.2.3 Proof-of-Work of the LDPC decoder

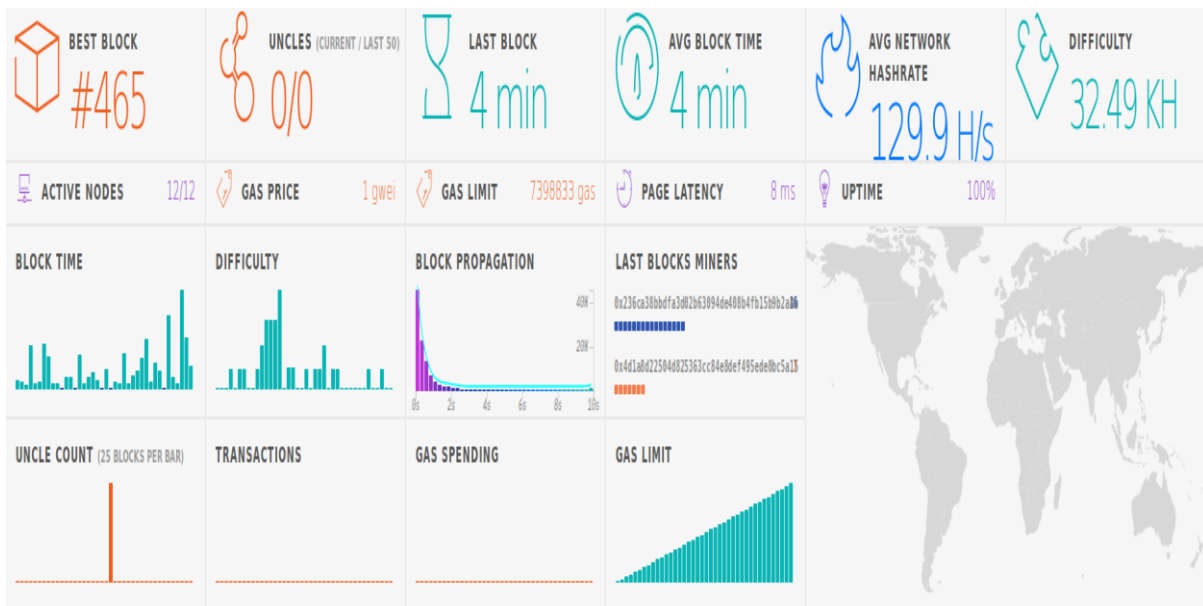
The goal of the LDPC decoder is to find a hash vector  $\hat{\mathbf{c}}$  that satisfies (1). The below definition explains decoding steps.

**Definition 2.** When PCM  $\mathbf{H}$ , and hash vector  $\mathbf{r}$  are given, the LDPC decoder uses  $\mathbf{H}$  and  $\mathbf{r}$  as inputs and obtains output  $\hat{\mathbf{c}}$  using message-passing algorithm [8], [20]. When  $\hat{\mathbf{c}}$  satisfies (1),  $\hat{\mathbf{c}}$  becomes an LDPC code and LDPC decoding is completed

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

A PCM  $\mathbf{H}$  is randomly generated, but all miners use the same previous hash value derived from the previous block. Thus, it is not possible to generate the next PCM to mine a block in advance. In the PoW of Bitcoin and Ethereum, nonce is changed when the result of hash function is smaller than target value. We follow similar way as that used by Ethereum to obtain a hash value with nonce. However, ECCPoW does not compare with target value but it checks result of decoding. When the code derived by (4) does not satisfy (1), node changes nonce and repeats all the steps.

Our method ECCPoW is based on the *high complexity of ASIC design* in Chapter 3 for an ASIC-resistant PoW. However, unlike previous researches, ECCPoW generate variable crypto puzzle by using continuously changing PCM and variable length code. It can be released ASIC of ECCPoW for specific length of code and PCM. However, it is possible to make ASIC obsolete by changing parameters. Furthermore, in [20], [21], it has been proven that implementing an ASIC that can handle variable PCM is expensive and occupies a lot of space. If buying CPU or GPU is more efficient, there is no reason to make an ASIC. In other words, the ECCPoW algorithm is ASIC resistant as implementing an ASIC

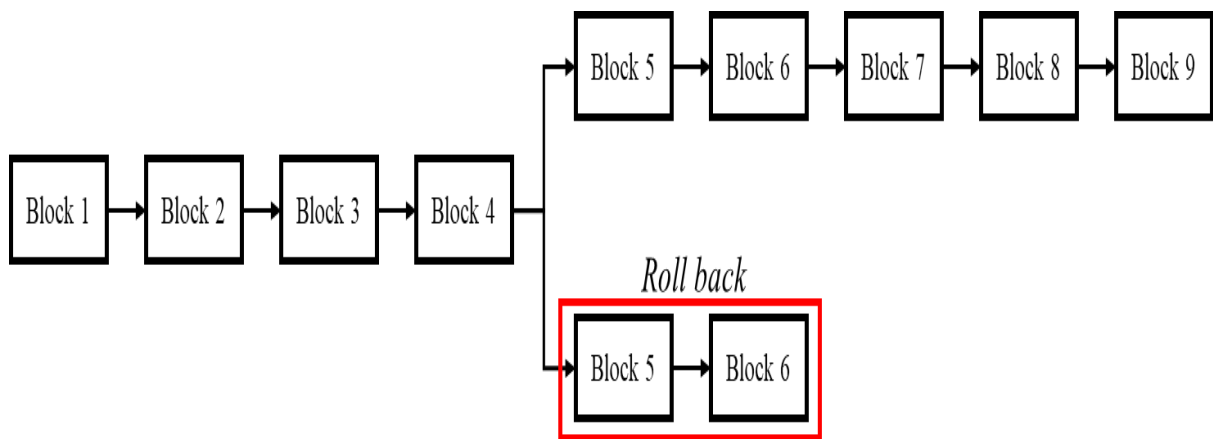


**Figure 13 Simulation of ECCPoW**

that can handle various lengths of changing codes and PCMs is very inefficient.

#### 4.2.4 Difficulty Control of ECCPoW Ethereum

In this subchapter, we give a comparison of difficulty implementation of Bitcoin and Ethereum. Also, we present the implementation of difficulty control of ECCPoW Ethereum. Bitcoin update difficulty every 2016 blocks. However, Ethereum update difficulty every block. Namely, in Bitcoin, even though a block is generated slower or faster than expected block generation time, the difficulty is not changed until a specific period. However, in Ethereum, the difficulty is changed without procrastination. Figure 8 and Figure 10 present practical examples. Figure 8 presents the difficulty control mechanism of Ethereum. It shows that Ethereum changes difficult every block. If the network generates a block with unexpected time, then the network changes difficulty. Ethereum allows generating a block within [9, 18) second. Thus, if the network generates a block with less than 9 seconds, then the network increases the difficulty. Figure 11 Difficulty control mechanism of Bitcoin unlike Ethereum, Bitcoin changes every 2016 blocks. The expected block generation time of Bitcoin is 10 minutes. When the network generates a block every 10 minutes, generating 2016 blocks take exactly two weeks. Thus, if generating the 2016 block takes more than two weeks, it means that the current difficulty is high for this network.



**Figure 14 Example case of rollback**

On the contrary, generating the 2016 block takes less than two weeks means the current difficulty is low for network.

Furthermore, in Pow base blockchain, blocks can be generated at the same time. Bitcoin and Ethereum consider it differently when network control difficulty. Bitcoin does not allow to generate a block at the same time. However, Ethereum allows three blocks to generate a block at the same time. Specifically, in Bitcoin, when blocks are generated at the same time, a block that is connected to the longest block is confirmed. Thus, the network rolls back transactions that are not included in the longest chain; it is called the longest chain rule. Figure 10 오류! 참조 원본을 찾을 수 없습니다. shows an example of the longest chain rule. In the longest chain rule, only a block that is connected to the previous can affect the difficulty. However, in Ethereum, the network allows three blocks. In these three blocks, only one block can include transactions. Blocks that cannot include transactions are called uncle block. Unlike Bitcoin that uses the longest chain rule, the Ethereum uses the heaviest chain rule. Figure 9 presents an example of the heaviest chain rule. Even though there exists the longest chain, the network selects the heaviest chain which includes more blocks. In the heaviest chain rule, uncle block can affect the difficulty. Namely, the network considers the hash rate to generate uncle blocks to control difficulty. Bitcoin does not consider uncle blocks to control blocks. It means that the network does not include a hash rate to generate an uncle block. Thus, even though the network has a high hash rate, the network



cannot fully utilize the hash rate for the security of the network. In PoW based blockchain, hash rate affects the security of blockchain. Because when the network has a low hash rate, a malicious miner can reverse blockchain using strong hash rate mining machine, such as ASIC or FPGA.

ECCPoW Bitcoin is based on Bitcoin [9]; ECCPoW Ethereum is based on Ethereum. Thus, ECCPoW Ethereum needs its own difficulty control unlike ECCPoW Bitcoin. For difficulty of ECCPoW Ethereum, we present how to apply error correction code process to difficulty control methods of Ethereum.

In [5], difficulty of Ethereum is defined by probability of block generation. It is defined as

$$n \leq \frac{2^{256}}{Diff} \quad (5)$$

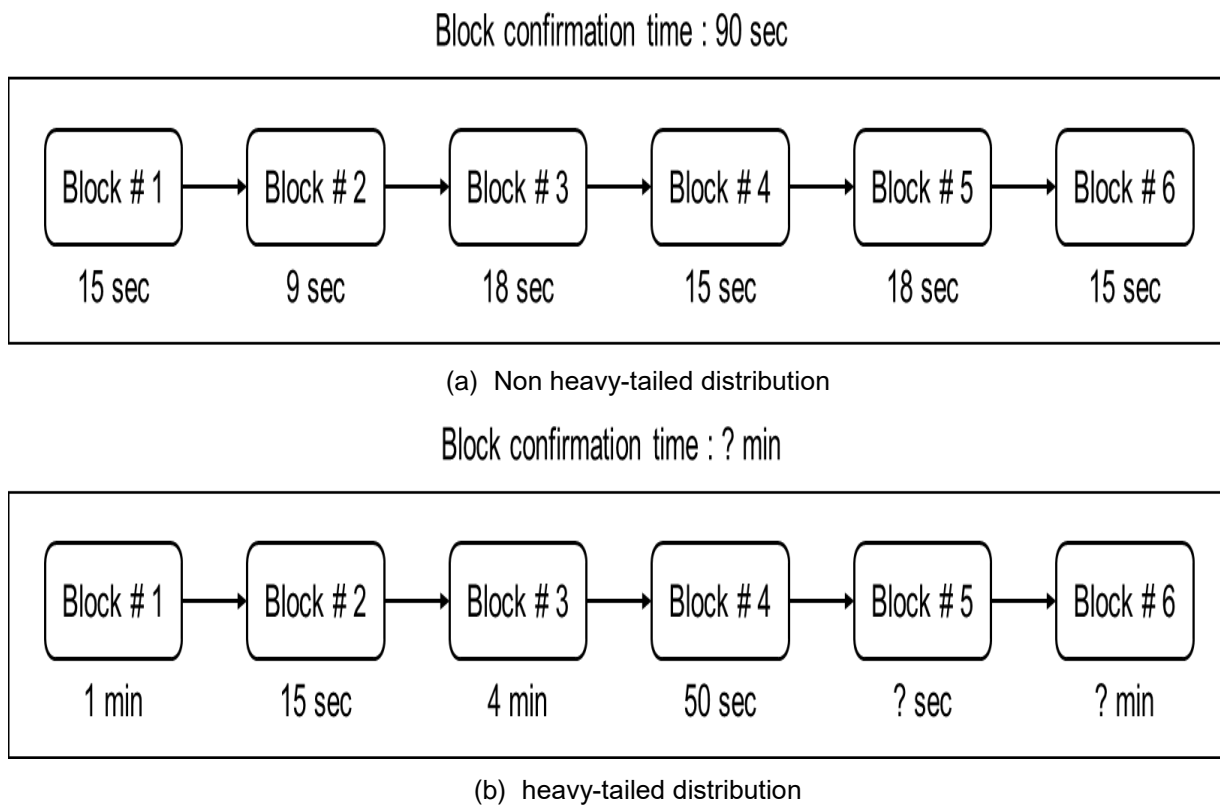
and indicates that

$$Diff \leq \frac{2^{256}}{n} \quad (6)$$

Where  $n$  denotes the result of PoW, and  $Diff$  denotes the difficulty of Ethereum. Thus, (6) means that when the difficulty increase, the value of  $n$  that satisfies (6) decrease. Furthermore, we can consider that the reciprocal of difficulty is a probability of block generation. Namely, if we can calculate a probability of block generation, it is possible to control difficulty similar to the process in Ethereum. Thus, it is important to know the probability of a successful LDPC decoding according to the LDPC parameter. To test the difficulty change using the block generation time, we calculated the pseudo-probability of a successful LDPC decoding according to the parameters. The parameters can be found at our GitHub<sup>2</sup>.

---

<sup>2</sup> [https://github.com/cryptoecc/ETH-ECC/blob/master/consensus/eccpow/LDPCDifficulty\\_utils.go#L65](https://github.com/cryptoecc/ETH-ECC/blob/master/consensus/eccpow/LDPCDifficulty_utils.go#L65)



**Figure 15 Problem of the heavy-tailed distribution**

In Figure 13 Simulation of ECCPoW, the difficulty of the ECCPoW algorithm is 32.49 KH, indicating that the probability of block generation is 1 of 32,490 hash cycles. Hash cycles denote the number of attempts to generate block. The difficulty of ECCPoW Ethereum is changed using a previous BGT, similar to that of Ethereum. If the last block is generated earlier than expected, the difficulty increases. Conversely, if the last block is generated later than expected, the difficulty decreases. Thus, if there are more hash cycles than expected, the difficulty increases, and when there are less hash cycles than expected, the difficulty decreases.

# Chapter 5

## 5 Problem Formulation

In PoW, there is a case where transactions are included in an orphan block which is not linked to chain. Transactions in an orphan block are rolled back; Figure 13 shows a case of roll back. Therefore, in PoW, the participants must wait for the block confirmation time to prevent transactions from rolling back. That is to say, in blockchains utilizing PoW, the BGT must have a finite mean for the block confirmation time. For example, if the BGT has an infinite mean, we cannot determine how long we must wait for the confirmation of transactions. Therefore, to apply the ECCPoW algorithm in a real network, the BGT must have a finite mean.

In [8], a theorem about the block generation of the ECCPoW algorithm using a hash cycle with a geometric distribution is presented. However, it has been assumed that the block must be generated within specific hash cycles. However, if the BGT has an infinite mean, there is no guarantee that the block will be generated within specific hash cycles. In [9], the practical experiment using the ECCPoW algorithm has been discussed. However, it was only mentioned that the ECCPoW algorithm has an unstable BGT, and no discussion on the mean of BGT was present. In this paper, the experimental result present moderate evidence that exponential distribution describes distribution of BGT of the ECCPoW algorithm with its implementation on Ethereum. Figure 14 presents problem of the heavy-tailed distribution. (a) shows non heavy-tailed distribution case, in this case, network can guarantee block confirmation time. However, in (b), it has high difference of block generation between blocks. Thus, the network cannot guarantee block confirmation time. Namely, if BGT of ECCPoW follows heavy-tailed distribution, it will show BGT such as (b) of Figure 14. Thus, to use ECCPoW in the real world, it is important to show that BGT of ECCPoW follows non heavy-tailed distribution.

# Chapter 6

## 6 Experiment On ECCPoW Ethereum

In this section, we conduct experiments using ECCPoW Ethereum. First, we simulate the difficulty change using multinode networks. Second, we conduct a goodness-of-fit experiment using the Anderson-darling (AD) test [23], [24], [25] to confirm the stability of the BGT with fixed difficulty.

### 6.1 Experiment Set Up

In this subchapter, we present how to set up experiment. This subchapter is organized as follow:

- Present experiment set up ECCPoW Ethereum node on Linux.
- Present experiment set up ECCPoW Ethereum node on Windows.
- Present how to connect nodes.

In first two subchapters, we present setting up ECCPoW Ethereum using only one node; in the last subchapter, we present a way to connect each node. In this subchapter, we use two fonts to distinguish command on terminal and terminal on ECCPoW Ethereum client: geth.

```
Command on terminal
```

Above font presents command on terminal.

```
Command on geth
```

Above font presents command on geth

#### 6.1.1 Experiment set up on Linux

This Chapter is based on below environments

- Linux mint 19.1 or Linux manjaro 19.0.
- Golang (version 1.10 or later).

#### Download and install geth

In this subchapter we present how to download ECCPoW Ethereum client: geth. To download geth,

1. Move to a directory which you want to locate ECCPoW Ethereum.

2. Open terminal using right-click of mouse.
3. Paste below command.

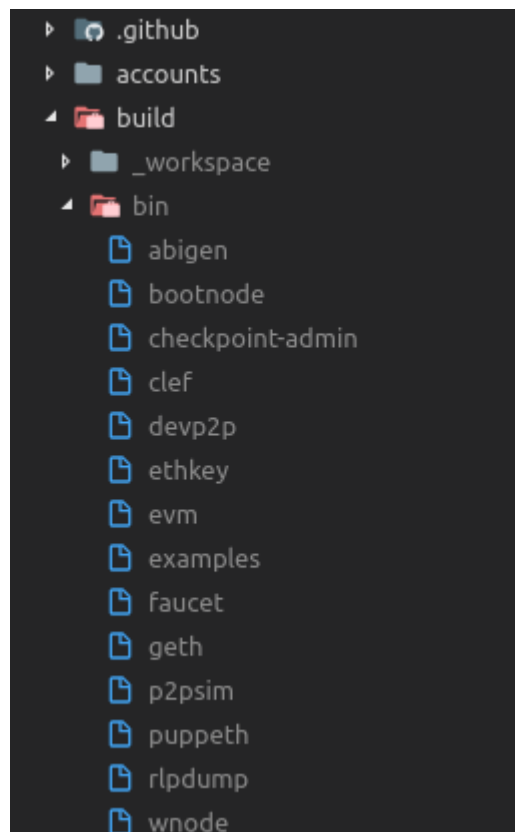
```
git clone https://github.com/cryptoecc/ETH-ECC
```

Then geth will be downloaded. After downloading,

1. Move to downloaded folder.
2. Open terminal using right-click and paste below line.

```
make all
```

If build is successful, then you can find binary files in bin folder like Figure 16.



**Figure 16 Binary files in bin folder**

## **Build**

Like we have mentioned, for this private network, we use only one node for private network. Connecting each node will be explained in the end of this subchapter 6.1. For building private network, we use puppeth. You can see puppeth in Figure 16. The puppeth is used to generate a genesis file. Before

build network, not only private but also public, we must generate a genesis file. The genesis file includes information of network. Thus, it is used to identify network. To run puppeth:

1. Move bin folder.
2. Open terminal using right-click of mouse and paste below line.

```
./puppeth
```

3. After run the puppeth, you can see a welcome message. Follow below steps.

```
+-----+
| Welcome to puppeth, your Ethereum private network manager |
| |
| This tool lets you create a new Ethereum network down to |
| the genesis block, bootnodes, miners and ethstats servers |
| without the hassle that it would normally entail.         |
| |
| Puppeth uses SSH to dial in to remote servers, and builds |
| its network components out of Docker containers using the |
| docker-compose toolset.                                   |
+-----+

Please specify a network name to administer (no spaces, hyphens or capital letters please)
> eccpow1

Sweet, you can set this via --network=eccpow1 next time!

INFO [02-14|21:21:23.414] Administering Ethereum network          name=eccpow1
WARN [02-14|21:21:23.414] No previous configurations found
path=/home/hskim/.puppeth/eccpow1

What would you like to do? (default = stats)
  1. Show network stats
  2. Configure new genesis
  3. Track new remote server
  4. Deploy network components
> 2

What would you like to do? (default = create)
  1. Create new genesis from scratch
  2. Import already existing genesis
> 1

Which consensus engine to use? (default = clique)
  1. Ethash - proof-of-work
  2. Clique - proof-of-authority
```

```

3. EccPoW - proof-of-work with LDPC
> 3

Which accounts should be pre-funded? (advisable at least one)
> 0x

Should the precompile-addresses (0x1 .. 0xff) be pre-funded with 1 wei? (advisable yes)
>

Specify your chain/network ID if you want an explicit one (default = random)
> 12345
INFO [02-14|21:21:58.917] Configured new genesis block

What would you like to do? (default = stats)
  1. Show network stats
  2. Manage existing genesis
  3. Track new remote server
  4. Deploy network components
> 2

  1. Modify existing configurations
  2. Export genesis configurations
  3. Remove genesis configuration
> 2

Which folder to save the genesis specs into? (default = current)
  Will create eccpow1.json, eccpow1-aleth.json, eccpow1-harmony.json, eccpow1-parity.json
>
INFO [02-14|21:22:02.800] Saved native genesis chain spec      path=eccpow1.json
ERROR[02-14|21:22:02.800] Failed to create Aleth chain spec      err="unsupported consensus engine"
ERROR[02-14|21:22:02.800] Failed to create Parity chain spec     err="unsupported consensus engine"
INFO [02-14|21:22:02.803] Saved genesis chain spec                client=harmony path=eccpow1-harmony.json

What would you like to do? (default = stats)
  1. Show network stats
  2. Manage existing genesis
  3. Track new remote server
  4. Deploy network components
> ^C

```

^C in the end of above step means ctrl + C.

We set a name as eccpow1. In the end of below step, eccpow1.json will be located in directory. Also, we set a chain/network ID as 12345 for convenient. You can change chained to any 256-bit number

except already occupied chains such as 1 of Ethereum. Because geth recognizes a network using chain/network ID.

We finished set up to run geth. Additionally, we have to make a storage directory to store data such as blocks and states of ECCPoW Ethereum. You can generate a directory to any location. In our case, location of storage directory is:

```
/home/hskim/Documents/geth-test
```

And name of storage directory is geth-test. Blocks and states will be stored in storage directory.

### Set up geth

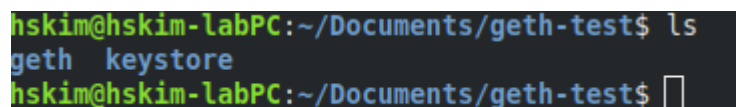
We installed geth and set directory to store data. In this subchapter, we present how to run it. Before run geth, move bin folder which is shown in Figure 16, open terminal and paste below command with replacement to initialize storage directory.

```
./geth --datadir {your_own_storage_directory} init {genesis_file_name.json}
```

For example,

```
./geth --datadir /home/hskim/Documents/geth-test init eccpow1.json
```

Location of our storage directory is /home/hskim/Documents/geth-test and name of genesis file is eccpow1.json. Thus, we replaced using ours. If above command works well, then you will be able to check new folders on your storage directory like Figure 17.



```
hskim@hskim-labPC:~/Documents/geth-test$ ls
geth keystore
hskim@hskim-labPC:~/Documents/geth-test$
```

**Figure 17 Result of storage directory initialization**

All of preparation to run private network is done. Now we show how to run private network. Open the terminal on bin folder which are described in Figure 15 and paste below command with replacement.

```
./geth --datadir Your_own_storage --networkid network_id console
```



For example,

```
./geth --datadir /home/hskim/Documents/geth-test --networkid 12345 console
```

If you did not set chain/network ID as 12345, then you have to replace it to your own chain/network ID.

After running above command, you can see below message.

```
INFO [08-06|21:27:43.867] Maximum peer count          ETH=50 LES=0
total=50
INFO [08-06|21:27:43.867] Smartcard socket not found, disabling  err="stat
/run/pcscd/pcscd.comm: no such file or directory"
INFO [08-06|21:27:43.870] Starting peer-to-peer node          in-
stance=Geth/v1.9.2-unstable-aa6005b4-20190805/linux-amd64/go1.12.7
INFO [08-06|21:27:43.870] Allocated trie memory caches
clean=256.00MiB dirty=256.00MiB
INFO [08-06|21:27:43.870] Allocated cache and file handles      data-
base=/home/hskim/Documents/geth-test/geth/chaindata cache=512.00MiB han-
dles=524288
INFO [08-06|21:27:43.904] Opened ancient database              data-
base=/home/hskim/Documents/geth-test/geth/chaindata/ancient
INFO [08-06|21:27:43.904] Initialised chain configuration      con-
fig="{ChainID: 12345 Homestead: 0 DAO: <nil> DAOSupport: false EIP150: <nil>
EIP155: 0 EIP158: 0 Byzantium: <nil> Constantinople: <nil> Petersburg: <nil>
Engine: unknown}"
INFO [08-06|21:27:43.904] Disk storage enabled for ethash caches
dir=/home/hskim/Documents/geth-test/geth/ethash count=3
INFO [08-06|21:27:43.904] Disk storage enabled for ethash DAGs
dir=/home/hskim/.ethash count=2
INFO [08-06|21:27:43.904] Initialising Ethereum protocol      versions=[63]
network=12345 dbversion=7
INFO [08-06|21:27:43.944] Loaded most recent local header      number=0
hash=ab944c...55600c td=400 age=50y3mo3w
INFO [08-06|21:27:43.944] Loaded most recent local full block  number=0
hash=ab944c...55600c td=400 age=50y3mo3w
INFO [08-06|21:27:43.944] Loaded most recent local fast block  number=0
hash=ab944c...55600c td=400 age=50y3mo3w
INFO [08-06|21:27:43.945] Loaded local transaction journal     transac-
tions=0 dropped=0
INFO [08-06|21:27:43.945] Regenerated local transaction journal transactions=0 accounts=0
INFO [08-06|21:27:43.951] Allocated fast sync bloom           size=512.00MiB
INFO [08-06|21:27:43.951] Initialized fast sync bloom         items=0 er-
rorrate=0.000 elapsed=37.353µs
INFO [08-06|21:27:43.997] New local node record               seq=3
id=65c5b16ab4aa9e9f ip=127.0.0.1 udp=30303 tcp=30303
INFO [08-06|21:27:43.998] Started P2P networking
self=enode://3e6e6cc9fd56954e02f3807813e086827ddf0576d0c969f67a915691ec3f8798673
32ba4911048fd513672856c63a2746063706005c6d777f670ae16c2c4a384@127.0.0.1:30303
```

```

INFO [08-06|21:27:43.999] IPC endpoint opened
url=/home/hskim/Documents/geth-test/geth.ipc
WARN [08-06|21:27:44.088] Served eth_coinbase reqid=3
t=16.874µs err="etherbase must be explicitly specified"
Welcome to the Geth JavaScript console!

instance: Geth/v1.9.2-unstable-aa6005b4-20190805/linux-amd64/go1.12.7
at block: 0 (Thu, 01 Jan 1970 09:00:00 KST)
datadir: /home/hskim/Documents/geth-test
modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0
rpc:1.0 txpool:1.0 web3:1.0
>

```

Now we are ready to run ECCPoW Ethereum.

### Run private network on Linux

Now we can run our own network. In this Chapter, we show how to make account, start mining, and make a transaction. Before start, we can check current block length and accounts of this network.

```

> eth.blockNumber
0
> eth.accounts
[]

```

On geth, type `eth.blockNumber` and `eth.account` separately. Then you can see above result. These results shows that currently, there is no block and account. We can generate an account using command on `geth`.

```

> personal.newAccount("Alice")
INFO [08-06|21:33:36.241] Your new key was generated address=0xb8c941069cc2b71b1a00db15e6e00a200d387039
WARN [08-06|21:33:36.241] Please backup your key file!
path=/home/hskim/Documents/geth-test/keystore/UTC--2019-08-06T12-33-34.442823142Z--b8c941069cc2b71b1a00db15e6e00a200d387039
WARN [08-06|21:33:36.241] Please remember your password!
"0xb8c941069cc2b71b1a00db15e6e00a200d387039"

```

We generate an account of “Alice” by using command `personal.newAccount(“Alice”)`. As a result of generation, we got the address of Alice: `"0xb8c941069cc2b71b1a00db15e6e00a200d387039"`. If we check accounts of network again,

```

> eth.accounts
["0xb8c941069cc2b71b1a00db15e6e00a200d387039"]

```

The geth shows that there is account of Alice. We will use Alice's address as a miner's address. If Alice success to generate a block, then ether will be send to Alice's account. Before mining, we need to check balance of Alice's account.

```
> eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
0
```

It shows there is no balance in Alice's account.

### **ECC-ETH mining**

Before mining, we have to set miner's account for coinbase transaction. When miner success to generate a block, then reward will be sent to miner's account. It is called as coinbase transition. Thus, we have to allocate an account to miner. For allocation, we use `miner.setEtherbase("account")` command.

```
> miner.setEtherbase("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
True
```

If account exists, then it shows True. After getting True, we can start mining. For mining, we use `miner.start("number of threads")`. For test, we use one thread. Thus, "number of threads" is replaced to 1.

```
> miner.start(1)
null
INFO [08-06|21:42:38.198] Updated mining threads          threads=1
INFO [08-06|21:42:38.198] Transaction pool price threshold updated
price=1000000000
null
> INFO [08-06|21:42:38.198] Commit new mining work          number=1
sealhash=4bb421...3f463a uncles=0 txs=0 gas=0 fees=0 elapsed=325.066µs
INFO [08-06|21:42:40.752] Successfully sealed new block          number=1
sealhash=4bb421...3f463a hash=4b2b78...4808f6 elapsed=2.554s
INFO [08-06|21:42:40.752] ⚡ mined potential block          number=1
hash=4b2b78...4808f6
.
.
.
INFO [08-06|21:42:56.174] ⚡ mined potential block          number=9
hash=2faebb...8be693
INFO [08-06|21:42:56.174] Commit new mining work          number=10
sealhash=384aa6...cb0596 uncles=0 txs=0 gas=0 fees=0 elapsed=179.463µs
> miner.stop()
null
```

We mined 9 blocks and stopped using `miner.stop()` command. We can check the number of blocks in

network:

```
> eth.blockNumber
9
```

It shows that we mined 9 block as we have known. Also, we can check result of coinbase transactions:

```
> web3.fromWei(eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039"),
"ether")
45
```

It shows that in account of Alice, there are 45 Ether. Because in source code, reward of coinbase transaction is defined as 5 Ether.

### Make a transaction

In this subchapter, we show how to send a ether to other account. For this, we generate one more account.

```
> personal.newAccount("Bob")
"0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90"
```

We will show sending ether from Alice's account to Bob's account. To send ether from Alice, we must unlock account of Alice first:

```
> web3.personal.unlockAccount("0xb8c941069cc2b71b1a00db15e6e00a200d387039")
Unlock account 0xb8c941069cc2b71b1a00db15e6e00a200d387039
```

Above command shows that Alice's account is unlocked. Now we can send ether to Bob's account. We will define two variables "Alice" and "Bob" for convenient. The "Alice" is defined as Alice's account and the "Bob" is defined as Bob's account.

```
> Alice = "0xb8c941069cc2b71b1a00db15e6e00a200d387039"
> Bob = "0xb8c941069cc2b71b1a00db15e6e00a200d387039"
```

We use these two variable to make a transaction.

```
> eth.sendTransaction({from: Alice, to: Bob, value: web3.toWei(5, "ether")})
```

Above command means Alice will send 5 ether to Bob's account. For confirmation of this transaction, we must wait until block is generated. In other word, we can see pending transaction which is waiting for generating a block.

```
> eth.pendingTransactions
[ {
  blockHash: null,
  blockNumber: null,
```

```

    from: "0xb8c941069cc2b71b1a00db15e6e00a200d387039",
    gas: 21000,
    gasPrice: 1000000000,
    hash: "0x926f1bb71d5b48a306e6cde2d45c01f8af2107febf94b166a7e5f8e025dc8adc",
    input: "0x",
    nonce: 0,
    r: "0x70484271bdc85f7233e715423d8d0be5c669a323385b5ec0f080a52cf3c654c",
    s: "0x1b55a792995f61128c10a48ce1e0869893c863d38489f574d84ae3a96b031cef",
    to: "0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90",
    transactionIndex: null,
    v: "0x42",
    value: 5000000000000000000
  }
}

```

Above lines show information of transaction. To confirm above transaction, we must generate a block.

If we generate a one block, we can see the change of balance in accounts.

```

> web3.fromWei(eth.getBalance("0xb8c941069cc2b71b1a00db15e6e00a200d387039"),
"ether")
45
> web3.fromWei(eth.getBalance("0xf39cf42cd233261cd2b45adf8fb1e5a1e61a6f90"),
"ether")
5

```

It shows that Alice sent 5 Ether to Bob but Alice get 5 Ether again from coinbase transaction.

### 6.1.2 Experiment set up on Windows

In this subchapter we present how to build ECC-ETH private network on Windows. Basically, ECC-ETH is considered to run on Linux. Therefore, we use virtual machine on Windows. Thus, we present how to install virtual machine and Linux Ubuntu.

#### Download VMware and Linux Ubuntu

In this subchapter, we present how to install VMware and Linux Ubuntu. To install download VMware, go to VMware download page<sup>3</sup> and for Linux Ubuntu, go to Ubuntu download page<sup>4</sup> and get a LTS version. After finishing download, install VMware and run it. The VMware will show you Figure 17. You can install Ubuntu by using “Create a New Virtual Machine”. After clicking it, you can see “Browse” tab like Figure 13. After setting a location of Ubuntu, click next and follow instruction to install.

<sup>3</sup> <https://www.vmware.com/kr/products/workstation-player/workstation-player-evaluation.html>

<sup>4</sup> <https://ubuntu.com/download/desktop>

# Welcome to VMware Workstation 15 Player



## Create a New Virtual Machine

Create a new virtual machine, which will then be added to the top of your library.



## Open a Virtual Machine

Open an existing virtual machine, which will then be added to the top of your library.



## Upgrade to VMware Workstation Pro

Get advanced features such as snapshots, virtual network management, and more.



## Help

View online help.

Figure 18 VMware

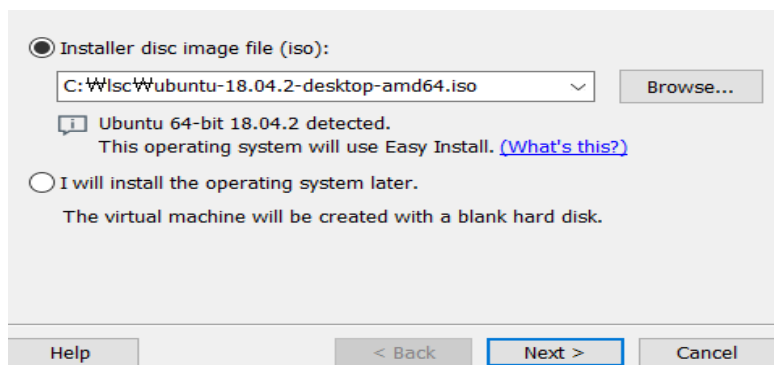


Figure 19 Step to install Ubuntu

```
infonet@ubuntu:~$ sudo apt-get update
```

**Figure 20 Update package**

```
infonet@ubuntu:~$ sudo apt-get install golang
```

**Figure 21 Install golang**

```
infonet@ubuntu:~$ sudo apt-get install git
```

**Figure 22 Install git**

### **Setting environment on Linux**

After finishing install, we must install packages to run ECC-ETH: update package, install golang, and git. We must install golang to compile ETH-ECC. Figure 19 shows command to update packages, Figure 20 shows command to install golang, and Figure 21 shows command to install git. After finishing above step, then next steps are same with subchapter 6.1.1.

### **6.1.3 How to connect nodes**

Before this subchapter we tested only using one node. In this subchapter, we present to connect nodes. To connect nodes, node must have a same genesis file. Thus in this subchapter, we consider each node has same genesis file.

### **Run bootnode**

First we run bootnode that will be connected with other nodes; we initialized geth on Chapter 6.1.1. To connect with other node, we run geth with IP address.

```
./geth --datadir /home/hskim/Documents/geth-test --networkid 12345 --nat extip:
172.**.**.* console
```

For this example, we use same storage directory and chain/network ID of Chapter 6.1.1. Replace “172.\*\*.\*\*.\*” to node’s IP address. After running above command, we have to get a nodeinfo of this node using `admin.nodeInfo.enode` command.

```
> admin.nodeInfo.enr
{"bootstrap-node-record"}
```

In our case:

```
> admin.nodeInfo.enr
"enr:-Je4QHb0h_OAfBhOgHG5Gqb6pc3hC1wpRvoNgh6yegYTAX-
PaShdjrE5dXqZZxG_xdi0_j3mYy9aqF0oiLzAyH1_tUcsBg2V0aMfGhHHG-
zwGAgmIkgnY0gmlwhKwaEHjC2VjcDI1NmsxoQJIB1U_j59OzT5P2wfFX-9-
XV6GYgx008AmXNUc62C5yoNOY3CCdmGDdWRwgnZh"
```

We use above information to connect with other node.

### Connect member node

For member node, we must initialize it using genesis file which is same with bootnode’s. After initialization using genesis file,

```
./geth --datadir {datadir} --networkid {12345} --bootnodes {bootstrap-node-rec-
ord} console
```

Replace `datadir` to member node’s storage directory and replace `bootstrap-node-record` to `admin.nodeInfo.enode` of bootnode. For example:

```
./geth --datadir ~/data --networkid 12345 --port 30305 --bootnodes "enr:-
Je4QHb0h_OAfBhOgHG5Gqb6pc3hC1wpRvoNgh6yegYTAX-
PaShdjrE5dXqZZxG_xdi0_j3mYy9aqF0oiLzAyH1_tUcsBg2V0aMfGhHHG-
zwGAgmIkgnY0gmlwhKwaEHjC2VjcDI1NmsxoQJIB1U_j59OzT5P2wfFX-9-
XV6GYgx008AmXNUc62C5yoNOY3CCdmGDdWRwgnZh" consol
```

You can check result of connection by command on `geth`.

```
> admin.peers
```



```

> admin.peers
[{"
  caps: ["eth/63", "eth/64", "eth/65"],
  enode: "enode://c7849d2a34705a2de48b74fd71f2d44cd579feb2e0cd0ec58d11db6b7d9c65b5be57ede1ccc7fa693d726d7ec687545538156054988bc0be9ba801ddf05f9bb0@192.248.149.212:30303",
  enr: "enr:-Jq4QAJJsCR_b90JCzDrV97HZB2BcE18vew97rZ71r2BKPV8EG0q1UBRUcY7VbR8_TAm_-GXfUzUVNl0RyYkV9nGp0EMg2V0aMfJhPxk7ASDEYwwgmLkgnY0gmlwhMD4ld5Jc2VjcdI1NmsxoQLHhJ0qNHBaLeSLdP1x8tRM1Xn-suDNDsWNEdtrfZxltYN0Y3CCdL-DdWRwgnZf",
  id: "741a3255336523a95f52e1d10432ff9a6d91b50629f577dfd7930cf3263bde33",
  name: "Geth/v1.9.23-stable-8c2f2715/linux-amd64/go1.13",
  network: {

```

## 6.2 Simulation of The Difficulty Change

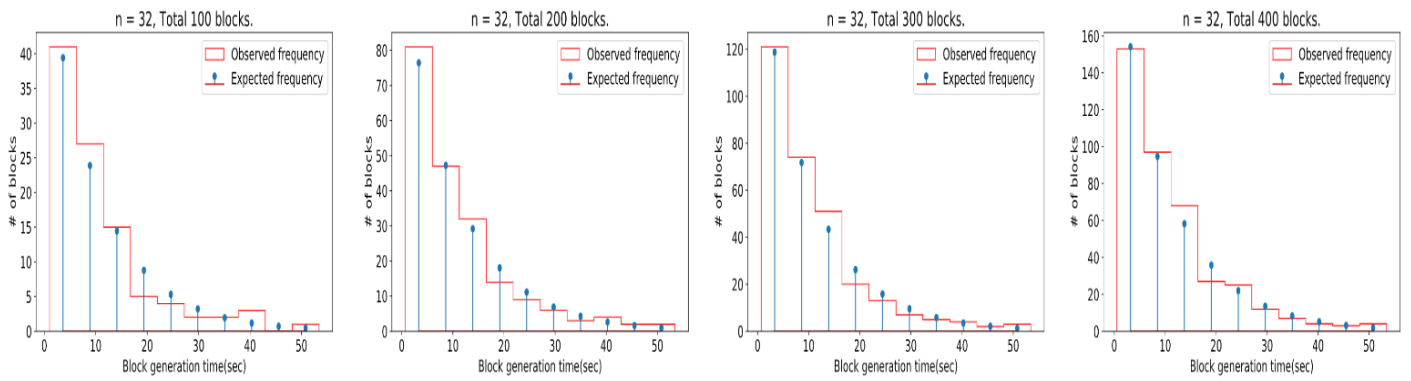
We simulate the difficulty change employing Amazon Web Services (AWS) using 12 nodes. Two nodes are *bootnodes* that help connect the nodes, and the other 10 nodes are *sealnodes* that participate in both the block validation and generation. In the charts presented in Figure 8, *BLOCK TIME* shows the BGT of the last 40 blocks, and *DIFFICULTY* shows the difficulty of the last 40 generated block. *BLOCK TIME* and *DIFFICULTY* show that because of large standard deviation, the block is generated fast despite the high level of difficulty, as already mentioned in [9]. In the next subchapter, we discuss about the BGT and difficulty. In the charts presented in Figure 8, *LAST BLOCK* shows the BGT of the last block, and *AVG BLOCK TIME* shows the average of the BGT of all the blocks.

Moreover, *AVG NETWORK HASHRATE* shows the average rate of the attempts of all miners. It can be calculated as follows:

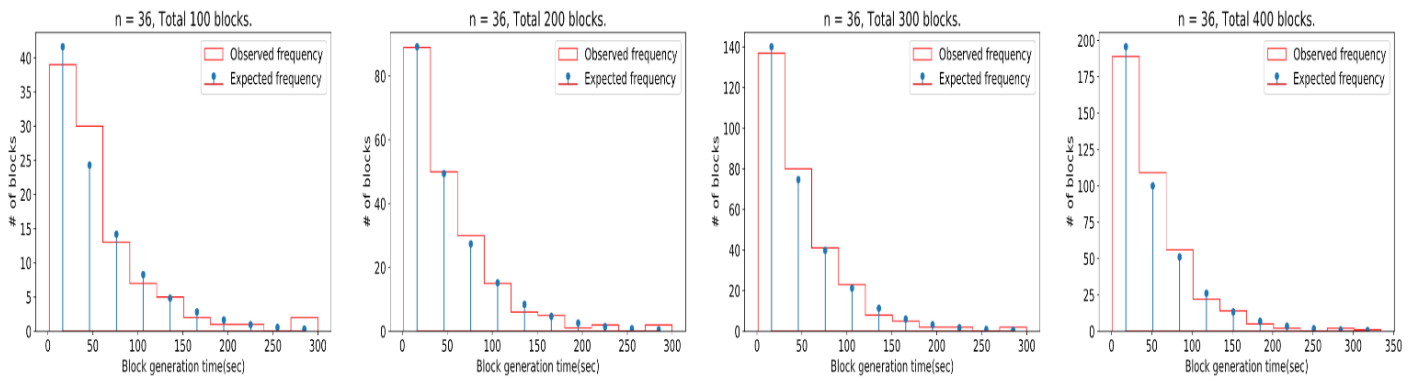
$$AVG_{HR} = \frac{Diff}{AVG_{BT}} \quad (7)$$

where  $AVG_{HR}$  denotes the *AVG NETWORK HASHRATE*;  $Diff$ , the *DIFFICULTY*, and  $AVG_{BT}$ , the *AVG BLOCK TIME*, which are presented in Figure 12.

*BLOCK PROPAGATION* shows the block propagation time from miner who generated block to other miners. We used two different regions: Seoul and US East. Specifically, 3 of the 10 *sealnodes* were in the US East region whereas the rest are in the Seoul region. *BLOCK PROPAGATION* also shows the percentage of blocks which are propagated corresponding times and indicates that it takes less than 2s



(a)  $n = 32$



(b)  $n = 36$

**Figure 24 Histogram of observed frequency and expected frequency**

to propagate between the Seoul and US East regions. Block propagation follows the same method as that of Ethereum.

### 6.3 Stability of The Block Generation Time

Figure 8 demonstrates the need to check if varying puzzles might make the BGT “unstable” which is mentioned in [9]. In *BLOCK TIME* and *DIFFICULTY* of Figure 8, a slow block generation can be observed despite the low level of difficulty. In other word, the observation of BGTs shows outliers. If the outliers are not controllable, the mean of the block generation is divergent similar to the heavy-tailed distribution. If the mean is none finite, the confirmation of transactions cannot be guaranteed. Thus, to achieve a stable BGT that can guarantee the confirmation of transactions, the BGT must have a finite mean.

We obtain the BGT of ECCPoW Ethereum with a fixed difficulty to observe BGT follows what kind of distribution. Specifically, if the distribution is exponential, the BGT is considered to have a finite mean. However, if the BGT follows a heavy-tailed distribution, it has an infinite mean [10]. Thus, through the simulation and goodness-of-fit, we aimed to determine what type of distribution the BGT follows. For the goodness-of-fit, we set a null hypothesis  $H_0$  and alternative hypothesis  $H_A$ :

$H_0$  : BGT has the exponential distribution

$H_A$  : BGT does not have the exponential distribution

We reject one of these hypotheses through the goodness-of-fit.

For the goodness-of-fit, we use the AD test [23], [24], [25]. However, the chi-squared test [26], Kolmogorov–Smirnov test [27], and AD test [23] can be considered for the goodness-of-fit. The chi-squared test has a restrictive assumption that all the expected frequencies should be 5 or more [28]. However, there is no guarantee that this can be achieved. If we collect more samples, the chi-squared test can be possibly used. However, the  $p$ -values used to validate the hypotheses are affected by the number of samples. When the number of samples is increased in the chi-squared test, the  $p$ -values tend to decrease. Therefore, the assumption of the chi-squared test is not appropriate for verifying our distributions. Unlike the chi-squared test, the Kolmogorov–Smirnov test does not depend on an adequate sample size. However, it is sensitive more to the center of the distribution rather than the tails [28]. To cover all possibilities, we must consider verifying the tail of distribution. Therefore, we use the AD test [23], which gives more weight to the tails compared with the Kolmogorov–Smirnov test.

### 6.3.1 Anderson-Darling test

The AD test is used to verify if a sample follows a specific distribution. We discuss one-sample and two-sample AD tests. In our work, we use the two-sample AD test; but to present our contribution clearly, we briefly introduce the one-sample AD test first. The one-sample test is suitable to verify 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 CDF of the empirical distribution is  $F_m(x)$ , the AD test [25] is used as follows:

$$A_m^2 = m \int_{-\infty}^{\infty} (F_m(x) - F(x))^2 w(x) dF(x) \quad (8)$$

and

$$w(x) = [F(x)(1 - F(x))]^{-1} \quad (9)$$

where  $m$  denotes the number of samples, and  $A_m^2$  denotes the results of the AD test. Intuitively, in (8), if  $F_m(x) - F(x)$  is 0 for all  $x$ ,  $A_m^2$  is 0. This indicates that when  $A_m^2$  is small, the empirical distribution  $F_m(x)$  is considered close to the population distribution  $F(x)$ . As we have noted, we aim to focus on the tail of the distribution which can be accomplished by Eq. (9). The one-sample AD test result  $A_m^2$  can be used to verify if the given sample comes from a population with a specific distribution.

In our work, we want to verify that two sample sets come from the same unknown population. The two-sample AD test is suitable for this verification. The two-sample AD test [24], [25] is as follows. There are two sample empirical distributions  $F_m(x)$  and  $G_n(x)$ . The  $F_m(x)$  is an empirical distribution made from the set  $\mathcal{F}$  with cardinality of samples set  $m = |\mathcal{F}|$ . The  $G_n(x)$  is also an empirical distribution made from the set  $\mathcal{G}$  with cardinality of samples set  $n = |\mathcal{G}|$ . For example,  $F_m(x)$  and  $G_n(x)$  are samples sets obtained independently from two different testing locations. It can be used to verify if the both sample distributions come from the same distribution. In [24], [25], the two-sample version is defined as follows:

$$A_{mn}^2 = \frac{mn}{K} \int_{-\infty}^{\infty} \frac{(F_m(x) - G_n(x))^2}{H_K(x)(1 - H_K(x))} dH_K(x) \quad (10)$$

**Table 1 Example of the Anderson-Darling test results**

The number of samples	Standardized $A_{mn}^2$	$p$ -value
10	-0.59	$p \geq 0.250$
20	0.44	$p = 0.21$
30	0.69	$p = 0.17$

(a)  $\mathcal{F} \sim \text{Exp}(1)$ ,  $\mathcal{G} \sim \text{Normal}(1,1)$

10	1.20	$p = 0.103$
20	3.57	$p = 0.011$
30	4.67	$p = 0.004$

(b)  $\mathcal{F} \sim \text{Exp}(1)$ ,  $\mathcal{G} \sim \text{Exp}(2)$

10	1.11	$p = 0.113$
20	-0.41	$p \geq 0.250$
30	-0.08	$p \geq 0.250$

(c)  $\mathcal{F} \sim \text{Exp}(1)$ ,  $\mathcal{G} \sim \text{Exp}(1)$

where  $H_K(x) = (mF_m(x) + nG_n(x)) / K$  with  $K = m + n$ .  $A_{mn}^2$  is standardized to remove the dependencies derived by the number of samples. This standardized form is utilized to calculate the  $p$ -value [24], [25]. The  $p$ -value provides an evidence for hypotheses test.

For the two-sample AD test, we set distributions  $F_m(x)$  and  $G_n(x)$ . have same population as a null hypothesis  $H_0$ . We set that the  $G_n(x)$  is an exponential distribution. Thus, if  $F_m(x)$  and  $G_n(x)$  comes from same population, namely  $H_0$  is true, we may consider that  $F_m(x)$  is the exponential distribution. If the  $p$ -value is large enough, the result of AD test provides the evidence that  $H_0$  is true.

The  $p$ -value is, under the assumption that the null hypothesis is true, the false-positive probability. A small  $p$ -value indicates that a test result provides evidence against the null hypothesis; a large  $p$ -value

does not. The  $p$ -value is determined from the observation of the sample data. Thus, before observing the data, we set the threshold significance level(TSL),  $TSL \in [0,1]$ , first. 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 [24] 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  $A_{mn}^2$  is smaller than the critical value, this result indicates that the  $p$ -value is higher than the predefined TSL. When the  $p$ -value is large, the probability of false-positive is large. In the significance level table of [24], the maximum TSL is 0.25. Thus, when standardized  $A_{mn}^2$  is lower than the critical value corresponding to the 0.25 TSL, the  $p$ -value is capped at 0.25.

In Table 1, we present three examples to give an insight of the  $p$ -value interpretation; in this example, we use true distributions for  $F_m(x)$  and  $G_n(x)$ . In Table 3,  $\text{Exp}(\theta)$  indicates the exponential distribution with mean  $\theta$  and  $\text{Normal}(\mu, \sigma)$  indicates the normal distribution with mean  $\mu$  and standard deviation  $\sigma$ ;  $m, n$  denotes the cardinality of sample sets that derived by true distributions. Namely,  $\mathcal{F} \sim \text{Exp}(\theta)$  denotes that the sample set  $\mathcal{F}$  has  $m$  samples; samples are derived from exponential distribution with mean  $\theta$ . In the (a) of Table 1, We use the exponential distribution for  $F_m(x)$  and the normal distribution for  $G_n(x)$ ; these distribution have same mean. This example presents, as the number of samples are increase,  $p$ -value tends to decrease if samples are drawn from different distributions, In the (b) of Table 1, we set the  $F_m(x)$  and the  $G_n(x)$  as the exponential distribution; but each distribution has different mean. This example presents, as the number of samples are increase, even though samples are drawn from the exponential distribution, the  $p$ -value tends to decrease if the means of distributions are different. In the (c) of Table, we set the  $F_m(x)$  and  $G_n(x)$  as the same exponential distribution.

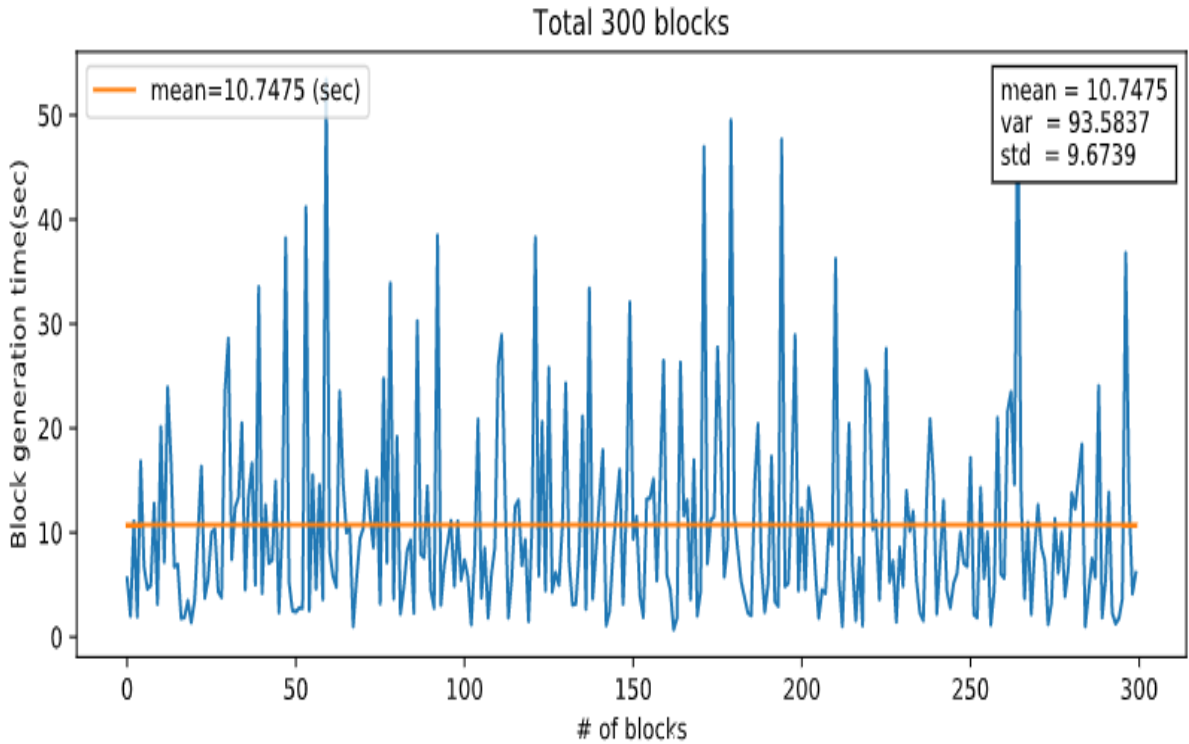
**Table 2 The observed frequency and expected frequency over time**

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

Namely,  $F_m(x)$  and  $G_n(x)$  have the same population. This example shows, as the number of samples increase, the  $p$ -value tends to increase if two samples are drawn from same population. By these examples of Table 1, we may consider that the closer the distribution of  $F_m(x)$  and  $G_n(x)$  are, the larger  $p$ -value is obtained.

We aim to see whether the AD test result of our experiments indicates that  $F_m(x)$  is close enough to  $G_n(x)$ . Namely, given there are two sets of samples, one derived by  $F_m(x)$  and the other derived by  $G_n(x)$  we want to see if the two sets of samples come, as a result of the AD test, from the same distribution. If the AD test result presents a significant  $p$ -value, i.e.,  $p \geq 0.25$ , it gives an enough evidence that  $F_m(x)$  is close to  $G_n(x)$  the exponential distribution.

### 6.3.2 Experimental detail



**Figure 25 Block generation time of 32 code length for 300 blocks**

For this experiment, 90 threads were used to generate a block. We experimented using a fixed code length to observe the BGT without difficulty change. 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 histogram. For example, when the minimum BGT is 10 and the maximum BGT is 20, there are [10,11], [11,12], ..., [19,20] intervals. Using these intervals, we classify the observed frequency of the BGT data in the histogram of blocks over interval. We set  $F_m(x)$  using the observed frequency. For the expected frequency in Table 2, the mean in Figure 25 is utilized to make the  $G_n(x)$  of Table 2. Namely, the mean in Figure 25 is used as  $1/\lambda$  for the CDF of the exponential distribution:

$$G_n(x) = 1 - e^{-\lambda x} \quad (11)$$



The expected frequency of the Table 1 is calculated using the integral in (11) corresponding to the interval. 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 the exponential distribution.

**Table 3 Anderson-Darling test results**

n	# of blocks	Observed mean(sec)	std	Standardized $A_{mn}^2$	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	$p \geq 0.25$

### 6.3.3 Experimental result

In Figure 24, we present a plot of the observed frequency and expected frequency. These frequencies are calculated in a manner mentioned in the Experiment detail. Figure 24 shows that the observed frequency tends to follow the expected frequency of the  $G_n(x)$ . Also, in Table 3, the observed mean and standard deviation tend to converge as the number of blocks increase. Furthermore, in Table 3, we present the results of the AD test to discuss hypotheses  $H_0$  and  $H_a$ . These results show a similar result of (c) of Table 3. In (c) of Table 3, we sample from the same true distribution; the results present the

high  $p$ -value. All the  $p$ -values in Table 3 have a higher than or equal to the 0.25 regardless of the number of blocks. In other words, if the null hypothesis is rejected, the probability of a false-positive is more than 25%. Therefore, we can conclude that result of the AD test presents moderate evidence; samples derived by  $F_m(x)$  and samples derived by  $G_n(x)$  come from close populations, so we may consider the distribution of  $F_m(x)$  follows the exponential distribution of  $G_n(x)$ .

# Chapter 7

## 7 Conclusion

In this paper, we present the implementation, simulation, and stability validation of ETH-ECC. In the implementation, we showed how Ethereum applies ECCPoW as a consensus algorithm with real implementation. In simulation, we conducted a multinode experiment using AWS EC2. The results revealed that the ECCPoW algorithm with varying difficulty is successfully implemented in the real world. In the stability validation, we showed the statistical results. These statistical results present moderate evidence that exponential distribution describes the distribution of block generation time of ECCPoW.

## REFERENCES

- [1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," [online]. Available: <https://bitcoin.org/bitcoin.pdf>
- [2] M. Rosenfeld, "Analysis of Hashrate-Based Double Spending," arXiv:1402.2009, Feb. 2014.
- [3] J. Jang and H.-N. Lee, "Profitable double-spending attacks," arXiv:1903.01711, Mar. 2019.
- [4] V. Buterin, "A next-generation smart contract and decentralized application platform," [online]. Available: <https://ethereum.org/en/whitepaper/>
- [5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Ethereum Project Yellow Paper, 2014.
- [6] E. Duffield, D. Diaz, "Dash:A payments-focused cryptocurrency," [Online]. Available: <https://github.com/dashpay/dash/wiki/Whitepaper>
- [7] T. Black, J. Weight "X16R ASIC Resistant by design," [Online]. Available: <https://ravencoin.org/assets/documents/X16R-Whitepaper.pdf>
- [8] S. Park, H. Choi, H. Lee, "Time-Variant Proof-of-Work Using Error Correction Codes," arXivL2006.12306, Jun. 2020.
- [9] H. Jung, H. Lee, "Error-Correction Code based Proof-of-Work for ASIC Resistance," Symmetry, 12, 988, Jun. 2020
- [10] S. Foss, D. Korshunov, S. Zachary, "An Introduction to Heavy-Tailed and Subexponential Distributions," Springer Science & Business Media, May. 2013.
- [11] M. Belotti, N. Božić, G. Pujolle and S. Secci, "A Vademecum on Blockchain Technologies: When, Which, and How," in IEEE Communications Surveys & Tutorials, vol. 21, no. 4, pp. 3796-3838, Fourthquarter. 2019.
- [12] L. Lamport, R. Shostak and M. Pease, "The byzantine generals' problem," ACM Trans. Program. Lang. Syst., vol. 4, no. 3, pp. 382–401,1982.
- [13] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," J.

- ACM, vol. 27, no. 2, pp. 228–234, 1980.
- [14] M. Castro and B. Liskov, “Practical Byzantine fault tolerance,” in Proc.OSDI, 1999, pp. 173–186.
- [15] E. Duffield, D. Diaz, "Dash:A payments-focused cryptocurrency," [Online]. Available: <https://github.com/dashpay/dash/wiki/Whitepaper>
- [16] T. Black, J Weight "X16R ASIC Resistant by design," [Online]. Available: <https://raven-coin.org/assets/documents/X16R-Whitepaper.pdf>
- [17] Tevador, "Random X, " [online]. Available: <https://github.com/tevador/RandomX>
- [18] T. Black, "Ravencoin — ASIC Thoughts — Round Two," [online]. Available: <https://medium.com/@tronblack/ravencoin-asic-thoughts-round-two-f4f743942656>
- [19] R. G. Gallager, "Low Density Parity Check Codes," Monograph M.I.T Press, 1963.
- [20] W. E. Ryan and S. Lin, Channel Codes Classical and modern, Cambridge, 2009.
- [21] S. Shao et al., "Survey of Turbo, LDPC, and Polar Decoder ASIC Implementations," in IEEE Communications Surveys & Tutorials, vol. 21, no. 3, pp. 2309-2333, third quarter 2019.
- [22] Y. Ueng, B. Yang, C. Yang, H. Lee and J. Yang, "An Efficient Multi Standard LDPC Decoder Design Using Hardware-Friendly Shuffled De-coding," in IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 60, no. 3, pp. 743-756, Mar. 2013
- [23] T. W. Anderson and D. A. Darling, "Asymptotic theory of certain ‘Goodness of Fit’ criteria based on stochastic processes," Ann. Math. Stat., vol. 23, no. 2, pp. 193-212, 1952.
- [24] A. N. Pettitt, "A two-sample Anderson–Darling rank statistic," Biometrika, vol. 63, no. 1, pp. 161-168, Apr. 1976.
- [25] F. Scholz and M. Stephens, "K-sample Anderson–Darling tests," Journal of the American Statistical Association, vol. 82, no. 399, pp. 918-924, 1987.
- [26] W. G. Cochran, "The  $\chi^2$  test of goodness of fit," The Annals of Mathematical Statistics, pp. 315-345, 1952.
- [27] J. L. Hodges, “The significance probability of the Smirnov two sample test,” Arkiv För

- Matematik, vol. 3, no. 5, pp. 469–486, Jan. 1958
- [28] W.G. Cochran, "Some Methods for Strengthening the Common Tests," *Biometrics*, vol. 10, pp. 417-451, 1954.
- [29] J.J. Filliben, "1.3.5.16. Kolmogorov-Smirnov Goodness-of-Fit Test," *NIST/SEMATECH e-Handbook of Statistical Methods*, [online]. Avail-able: <http://www.itl.nist.gov/div898/handbook/>

## ACKNOWLEDGEMENT

입학 했을 때에는 먼 미래라고 생각 했던 졸업이 매우 빠르게 다가온 것 같습니다. 2년 동안 교수님과 연구실 선후배님들께 정말 많은 것들을 배웠습니다. 너무나 많은 것들을 받은 것 같아 죄송스러운 마음입니다. 연구실에서 처음으로 회식을 할 때 자기소개로 했던 말이 아직도 머리 속에 남아있습니다. 그때, “연구실에 도움이 되는 사람이 되겠습니다.” 라고 말했었는데, 이 말을 지키기 위해 많은 시간을 연구실에서 보냈음에도, 받았던 것들이 훨씬 많은 것 같습니다. 2년간 많은 경험을 했고 즐거운 시간도 많았지만 아쉬움도 남습니다. 졸업 논문 제출을 앞둔 이 순간까지 작성중인 논문을 저널에 제출 못한 아쉬움이 마음속에 남아있습니다. 조금 더 빨리 논문을 쓰기 시작했더라면, 더 좋은 논문을 쓸 수 있었을 텐데 라는 아쉬움 역시 남아있습니다. 하지만 논문을 준비하면서 겪은 경험들은 미래의 저에게 큰 거름이 될 것이라고 믿습니다.

대학원을 졸업 할 때까지 너무나 많은 분들께 도움을 받은 것 같습니다. 항상 저를 사랑해주는 부모님과 우리 누나, 내가 가장 아끼는 강아지 대박이, 그리고 저에게 항상 많은 도움과 조언을 아끼지 않았던 교수님과 연구실 선후배님들께 진심으로 감사드립니다. 이분들의 앞날에 항상 행복한 일들만 있기를 기원하겠습니다. 정말 진심으로 감사합니다.