Proof of Luck: an Efficient Blockchain Consensus Protocol

https://youtu.be/XOT_HmBM4cM

개요

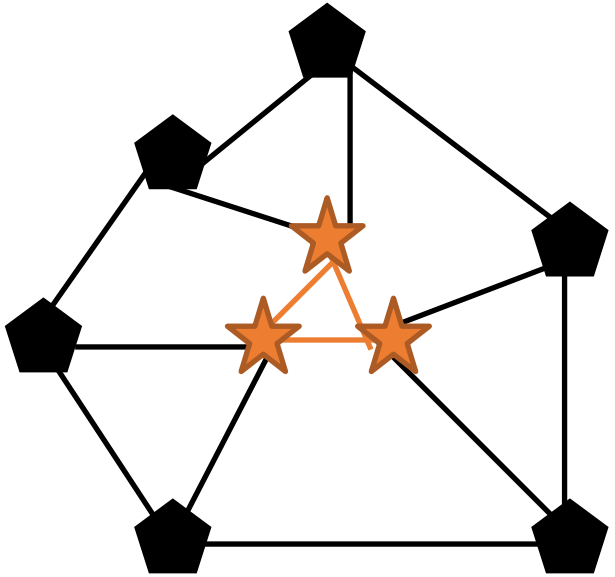
배경지식

TEE-based algorithms

TEE-based Proof-of-Luck

[개요]

- 기존의 합의 알고리즘에는 문제점이 존재
  - TEE를 통하여 기존의 합의 알고리즘을 발전시킨다.
    - PoW, PoT, PoO
    - TEE는 주요한 연산이 올바르게 동작하도록 할 수 있다.
    - Sybil attack에 대한 대책으로 사용될 수 있다.
    - ASIC에 대한 저항성을 가진다.



- Proof-of-Luck
  - 기존의 합의 알고리즘에 TEE를 적용한 방식에서 아이디어를 얻어 구현된 새로운 합의 알고리즘

[개요] 본 논문의 주요 목표 및 전제 조건

- 해당 디자인의 주요 목표는 다음과 같이 총 여섯 가지가 있습니다.
    1. 빠르고 결정론적인 transaction validation을 구현하는 것
    2. 네트워크 커뮤니케이션에 효율적인 프로토콜을 구축하는 것
    3. 커스텀 하드웨어인 ASIC에 대한 저항성을 갖도록하는 것
    4. TEE에 의해 공격자가 블록체인을 컨트롤할 수 없도록하는 것
    5. 공격자가 대부분의 CPU를 제거하거나 TEE 체계를 부수지 않는 이상 블록체인 네트워크를 컨트롤할 수 없도록하는 것
    6. 참가자들 사이의 synchronized된 clock에 대한 어떠한 요구사항도 존재하지 않도록하는 것

- 이를 위한 전제 조건은 다음과 같이 총 세 가지가 있습니다.
    1. 참가자는 Intel SGX와 같은 TEE를 실행하는 CPU를 가지고 있어야 한다.
    2. TEE는 공격자에 의해 영향을 받지 않는 uniform random number를 생성할 수 있어야 한다. (Intel SGX의 RDRAND)
    3. TEE 프로그램은 동시 호출을 탐지할 수 있다. (Intel SGX의 monotonic counter)

[배경지식] 네트워크 구성요소

- “*Participant*”: TEE를 사용해야 하며, 블록체인을 유지하고 다른 참가자들의 read, write를 돕기 위한 루틴을 수행한다. (채굴자)
- “*Client*”: 블록체인과 참가자들의 read, write에 의존한다. (일반 노드)
- “*Trusted platform vendor*”: 참가자들의 TEE내에 있는 알고리즘의 실행이 올바르게 돌아가도록 통제한다.

- 클라이언트는 참가자를 검증할 수 없으므로, 모든 데이터에 서명을 함으로써 블록체인의 내용을 보호한다.

[배경지식] TEE (Intel SGX)

- 본 논문은 Intel SGX를 기반으로 한다.
- 기존의 TEE의 2가지 attestation (isolation, remote)를 제공한다.
- Intel SGX만의 relative timestamp와 monotonic counter를 제공한다.
- 또한 anonymous, pseudonymous attestation을 사용하는 EPID(Enhanced Privacy ID)를 사용한다.

- TEE를 초기화할 때, platform은 TEE의 암호화 identity를 제공하는 코드 및 데이터의 해시값 (measurement)를 제공한다.
- Remote attestation을 통해 TEE는 서명된 “report"와 원격에서 검증가능한 “quote”의 계산 결과를 검증한다.
  - 이를 통하여 적절한 계산으로부터 결과값이 도출되었다는 것을 증명한다.

[TEE-based] Proof-of-Work

- PoW는 ASIC에 저항성을 가져야 한다.
- 이 때, TEE를 사용하여 이러한 문제를 해결할 수 있다.
- 결과적으로, 마이닝 파워가 탈 중앙화되며 mining pool의 이점이 사라진다.

Algorithm 1 TEE-enabled proof of work **(inside TEE)**

```
1: function PoW(nonce, difficulty)
2:     result ← originalPoW(nonce, difficulty)
3:     assert originalPoWSuccess(result)
4:     return TEE.attestation(⟨nonce, difficulty⟩, null)
5: end function
```

- Nonce: 새로 채굴된 블록의 헤더, difficulty: target hash
- null: anonymous random base EPID signature
- TEE.attestation -> 알고리즘이 TEE내에서 동작하는지, 코드가 수정되지 않았는지에 대해 measurement에 기반하여 검증하는 proof를 리턴한다.

[TEE-based] Proof-of-Time

- PoW는 참가자들에게 작업을 강요함으로써 일정 시간이 지나가도록 강제한다.
- TEE는 이러한 작업 과정 없이 원하는 시간만큼 기다리도록 요구할 수 있다.
- 이를 통해 CPU의 cycle과 자원을 절약하며, 그동안 다른 의미있는 작업을 수행하도록 할 수 있다.

Algorithm 2 Proof of time **(inside TEE)**

```
1: counter ← INCREMENTMONOTONICCOUNTER()
2: function POT(nonce, duration)
3:     SLEEP(duration)
4:     assert counter = READMONOTONICCOUNTER()
5:     return TEE.ATTESTATION(⟨nonce, duration⟩, null)
6: end function
```

- TEE의 relative timestamp를 사용한다.
- duration 기간동안 busy-wait를 하도록 하거나, 외부 프로세스에 제어를 양보한다.
- MonotonicCounter를 통해 병렬 실행을 하고자 하는 악의적인 참가자를 관리한다.

[TEE-based] Proof-of-Ownership

- PoW는 작업을 강요함으로써 Sybil attack에 저항성을 갖는다.
- TEE를 사용하게 된다면, 자원을 소모하는 것 대신 가상 참여자를 유지하는 것의 비용을 늘림으로써 물리적으로 제한할 수 있다.
- EPID를 통하여 동일한 CPU로부터 다중 attestation을 했는지에 대해서 드러내는 “pseudonym”을 생성한다.
- 이는 CPU의 owner epoch register를 리셋하더라도 동작한다.

Algorithm 3 Proof of ownership **(inside TEE)**

```
1: function PoO(nonce)
2:     return TEE.ATTESTATION(⟨nonce⟩, nonce)
3: end function
```

- Nonce: block header
- 유니크한 pseudonyms로 가장 많은 proof를 가진 블록을 리더로 선택함으로써 합의에 도달할 수 있다.

Proof-of-Luck

- 크게 PoLRound와 PoLMine으로 이루어져 있음

- PoLRound
  - 매 라운드를 시작할 때 PoLRound를 호출하며 가장 최근 block을 전달

- PoLMine
  - 새로운 블록의 header와 확장할 블록 (previousBlock)을 전달
  - previousBlock과 PoLRound의 roundBlock은 다르지만 parent는 동일
  - ROUND_TIME을 필수적으로 기다리도록 하여 (PoT)
  - 더 운이 좋은 블록이 존재할 경우 해당 블록으로 전환할 수 있도록 한다.
  - 이 경우 자신의 블록을 브로드캐스트할 필요가 없다.

  - Uniform distribution으로부터 [0, 1) 사이의 랜덤한 값을 생성한다.
  - 이를 통해 이번 라운드의 승리 블록을 결정한다.

Algorithm 4 Proof of luck primitive **(inside TEE)**

```
1:  counter ← incrementMonotonicCounter()
2:  roundBlock ← null
3:  roundTime ← null

4:  function PoLRound(block)
5:      roundBlock ← block
6:      roundTime ← tee.getTrustedTime()
7:  end function

8:  function PoLMine(header, previousBlock)
        // Validating link between header and previousBlock.
9:      assert header.parent = hash(previousBlock)
        // Validating previousBlock matches roundBlock.
10:     assert previousBlock.parent = roundBlock.parent
        // Validating the required time for a round passed.
11:     now ← tee.getTrustedTime()
12:     assert now ≥ roundTime + ROUND_TIME

13:     roundBlock ← null
14:     roundTime ← null
15:     l ← getRandom()
16:     sleep(f(l))

        // Validating that only one TEE is running.
17:     newCounter ← readMonotonicCounter()
18:     assert counter = newCounter

19:     nonce ← hash(header)
20:     return tee.attestation(⟨nonce, l⟩, null)
21: end function
```

Proof-of-Luck (Cont.)

- PoLMine
  - 프로토콜을 최적화하기 위해 f(l)을 통해 proof의 release를 지연시킨다.
  - 운이 좋은(더 큰)숫자에 대해 짧은 지연 시간을 부여하며
  - 운이 좋지 않은(더 작은) 숫자에 대해 긴 지연 시간을 부여한다.

  - Monotonic counter를 통해 동시 호출을 방지한다. (PoO)

Algorithm 4 Proof of luck primitive **(inside TEE)**

```
1:  counter ← incrementMonotonicCounter()
2:  roundBlock ← null
3:  roundTime ← null

4:  function PoLRound(block)
5:      roundBlock ← block
6:      roundTime ← tee.getTrustedTime()
7:  end function

8:  function PoLMine(header, previousBlock)
        // Validating link between header and previousBlock.
9:      assert header.parent = hash(previousBlock)
        // Validating previousBlock matches roundBlock.
10:     assert previousBlock.parent = roundBlock.parent
        // Validating the required time for a round passed.
11:     now ← tee.getTrustedTime()
12:     assert now ≥ roundTime + ROUND_TIME

13:     roundBlock ← null
14:     roundTime ← null
15:     l ← getRandom()
16:     sleep(f(l))

        // Validating that only one TEE is running.
17:     newCounter ← readMonotonicCounter()
18:     assert counter = newCounter

19:     nonce ← hash(header)
20:     return tee.attestation(⟨nonce, l⟩, null)
21: end function
```

Proof-of-Luck (Cont.)

```
179  function teeProofOfLuckRound(blockPayload) {
180    if (!verifyPayload(blockPayload)) {
181      throw new Error("Invalid blockPayload")
182    }
183
184    sleepCallback = null
185    sleepTime = null
186    roundBlockPayload = blockPayload
187    roundTime = teeGetTrustedTime()
188  }
```

```
function teeGetTrustedTime() {
  var trustedTime = SecureWorker.getTrustedTime()
```

Algorithm 4 Proof of luck primitive **(inside TEE)**

```
1:  counter ← INCREMENTMONOTONICCOUNTER()
2:  roundBlock ← null
3:  roundTime ← null

4:  function POLROUND(block)
5:      roundBlock ← block
6:      roundTime ← TEE.GETTRUSTEDTIME()
7:  end function

8:  function POLMINE(header, previousBlock)
        // Validating link between header and previousBlock.
9:      assert header.parent = HASH(previousBlock)
        // Validating previousBlock matches roundBlock.
10:     assert previousBlock.parent = roundBlock.parent
        // Validating the required time for a round passed.
11:     now ← TEE.GETTRUSTEDTIME()
12:     assert now ≥ roundTime + ROUND_TIME

13:     roundBlock ← null
14:     roundTime ← null
15:     l ← GETRANDOM()
16:     SLEEP(f(l))

        // Validating that only one TEE is running.
17:     newCounter ← READMONOTONICCOUNTER()
18:     assert counter = newCounter

19:     nonce ← HASH(header)
20:     return TEE.ATTESTATION(⟨nonce, l⟩, null)
21: end function
```

Proof-of-Luck (Cont.)

```
function teeProofOfLuckMine(payload, previousBlock, previousBlockPayload, callback) {
  if (previousBlock === null && previousBlockPayload === null) {
    teeProofOfLuckMineGenesis(payload, callback)
    return
  }

  if (sleepCallback !== null) {
    throw new Error("Invalid state, sleepCallback")
  }

  if (sleepTime !== null) {
    throw new Error("Invalid state, sleepTime")
  }

  if (roundBlockPayload === null || roundTime === null) {
    throw new Error("Invalid state, roundBlockPayload or roundTime")
  }

  if (!verifyPayload(payload)) {
    throw new Error("Invalid payload")
  }

  if (!verifyBlock(previousBlock)) {
    throw new Error("Invalid previousBlock")
  }

  if (!verifyPayload(previousBlockPayload)) {
    throw new Error("Invalid previousBlockPayload")
  }
```

Algorithm 4 Proof of luck primitive **(inside TEE)**

1: $counter \leftarrow$ INCREMENTMONOTONICCOUNTER()
2: $roundBlock \leftarrow$ **null**
3: $roundTime \leftarrow$ **null**

4: **function** POLROUND($block$)
5: $roundBlock \leftarrow block$
6: $roundTime \leftarrow$ TEE.GETTRUSTEDTIME()
7: **end function**

8: **function** POLMINE($header$, $previousBlock$)
// Validating link between *header* and *previousBlock*.
9: **assert** $header.parent =$ HASH($previousBlock$)
// Validating *previousBlock* matches *roundBlock*.
10: **assert** $previousBlock.parent = roundBlock.parent$
// Validating the required time for a round passed.
11: $now \leftarrow$ TEE.GETTRUSTEDTIME()
12: **assert** $now \geq roundTime + ROUND\_TIME$

13: $roundBlock \leftarrow$ **null**
14: $roundTime \leftarrow$ **null**
15: $l \leftarrow$ GETRANDOM()
16: SLEEP($f(l)$)

// Validating that only one TEE is running.
17: $newCounter \leftarrow$ READMONOTONICCOUNTER()
18: **assert** $counter = newCounter$

19: $nonce \leftarrow$ HASH($header$)
20: **return** TEE.ATTESTATION($\langle nonce, l \rangle$, **null**)
21: **end function**

Proof-of-Luck (Cont.)

```
230     ipfsHash(previousBlock).then(function (previousBlockIpfsHash) {
231       var payloadParentLink = getIpfsLink(payload, "parent")
232       if (!payloadParentLink || payloadParentLink.Hash !== previousBlockIpfsHash) {
233         throw new Error("payload.parent != hash(previousBlock)")
234       }
235
236       return ipfsHash(previousBlockPayload)
237     }).then(function (previousBlockPayloadIpfsHash) {
238       var previousBlockPayloadLink = getIpfsLink(previousBlock, "payload")
239       if (!previousBlockPayloadLink || previousBlockPayloadLink.Hash !== previousBlockPayloadIpfsHash) {
240         throw new Error("previousBlock.payload != hash(previousBlockPayload)")
241       }
```

Algorithm 4 Proof of luck primitive **(inside TEE)**

```
1:  counter ← INCREMENTMONOTONICCOUNTER()
2:  roundBlock ← null
3:  roundTime ← null

4:  function POLROUND(block)
5:      roundBlock ← block
6:      roundTime ← TEE.GETTRUSTEDTIME()
7:  end function

8:  function POLMINE(header, previousBlock)
        // Validating link between header and previousBlock.
9:      assert header.parent = HASH(previousBlock)
        // Validating previousBlock matches roundBlock.
10:     assert previousBlock.parent = roundBlock.parent
        // Validating the required time for a round passed.
11:     now ← TEE.GETTRUSTEDTIME()
12:     assert now ≥ roundTime + ROUND_TIME

13:     roundBlock ← null
14:     roundTime ← null
15:     l ← GETRANDOM()
16:     SLEEP(f(l))

        // Validating that only one TEE is running.
17:     newCounter ← READMONOTONICCOUNTER()
18:     assert counter = newCounter

19:     nonce ← HASH(header)
20:     return TEE.ATTESTATION(⟨nonce, l⟩, null)
21: end function
```

Proof-of-Luck (Cont.)

```
250       var now = teeGetTrustedTime()
251
252       if (now < roundTime + ROUND_TIME) {
253         throw new Error("now < roundTime + ROUND_TIME")
254       }
255
256       roundBlockPayload = null
257       roundTime = null
258
259       return ipfsHashBuffer(payload)
260     }).then(function (payloadIpfsHashBuffer) {
261       var payloadHash = new Uint8Array(payloadIpfsHashBuffer)
262       var l = teeGetRandom()
263
264       buildNonce(payloadHash, l, callback)
265     }).catch(function (error) {
266       callback(error)
267     })
268   }
```

Algorithm 4 Proof of luck primitive **(inside TEE)**

```
1:  counter ← INCREMENTMONOTONICCOUNTER()
2:  roundBlock ← null
3:  roundTime ← null

4:  function POLROUND(block)
5:      roundBlock ← block
6:      roundTime ← TEE.GETTRUSTEDTIME()
7:  end function

8:  function POLMINE(header, previousBlock)
        // Validating link between header and previousBlock.
9:      assert header.parent = HASH(previousBlock)
        // Validating previousBlock matches roundBlock.
10:     assert previousBlock.parent = roundBlock.parent
        // Validating the required time for a round passed.
11:     now ← TEE.GETTRUSTEDTIME()
12:     assert now ≥ roundTime + ROUND_TIME

13:     roundBlock ← null
14:     roundTime ← null
15:     l ← GETRANDOM()
16:     SLEEP(f(l))

        // Validating that only one TEE is running.
17:     newCounter ← READMONOTONICCOUNTER()
18:     assert counter = newCounter

19:     nonce ← HASH(header)
20:     return TEE.ATTESTATION(⟨nonce, l⟩, null)
21: end function
```

Proof-of-Luck (Cont.)

```
function teeGetRandom() {
  var array = new Uint32Array(2)
  crypto.getRandomValues(array)

  // Keep all 32 bits of the the first, top 20 of the second for 52 random bits.
  var mantissa = (array[0] * Math.pow(2, 20)) + (array[1] >>> 12)

  // Shift all 52 bits to the right of the decimal point.
  return mantissa * Math.pow(2, -52)
}
```

```
function f(l) {
  // We always wait at least one second. This allows all peers to at least compute their own lucky numbers
  // and then be able to know if they are winning or not, and if they should ignore less luckier blocks.
  return (1 - l) * ROUND_TIME / 2 + 1
}
```

Proof-of-Luck (Cont.)

- 참가자는 채굴한 블록의 체인이 기존의 체인보다 운이 좋을 경우 새 체인을 다른 참가자에게 브로드캐스트 해야 한다.
- 참가자는 newTransaction으로부터 생성된 newBlock을 포함한 새로운 체인을 리턴하기 위하여 commit을 사용한다.
- newBlock은 previous block의 hash인 parent, data인 newTransaction, 그리고 PoL로부터 생성된 proof를 포함한다.

Algorithm 5 Extending a blockchain with a new block

```
1: function COMMIT(newTransactions, chain)
2:     previousBlock ← LATESTBLOCK(chain)
3:     parent ← HASH(previousBlock)
4:     header ← ⟨parent, newTransactions⟩
5:     proof ← POLMINE(header, previousBlock)
6:     newBlock ← ⟨parent, newTransactions, proof⟩
7:     return APPEND(chain, newBlock)
8: end function
```

Proof-of-Luck (Cont.)

- Luck 알고리즘은 각 블록의 l value를 합하여 주어진 블록체인의 score인 luck을 계산한다.
- tee.proofData는 data가 proof을 생성할 때 사용되었는지 나타낸다.

Algorithm 6 Computing a luck of a valid blockchain

```
1: function LUCK(chain)
2:     luck ← 0
3:     for block in chain do
4:         luck ← luck + TEE.PROOFDATA(block.proof).l
5:     end for
6:     return luck
7: end function
```

- 네트워크가 나뉠 때, 더 큰 쪽이 더 큰 운을 갖게 된다.
- 즉, 소수의 공격자들이 포함된 체인은 다수의 참가자가 포함된 체인을 공격하지 못한다.

Proof-of-Luck (Cont.)

- valid 알고리즘은 첫 블록(genesis block)부터 가장 최근 블록까지 검증한다.
- 각 블록의 트랜잭션, PoL, 이전 블록의 해시가 일치하는지 등에 대해 검증한다.

Algorithm 7 Validating a blockchain

```
1:  function VALID(chain)
2:      previousBlock ← null
3:      while chain ≠ ε do
4:          block ← EARLIESTBLOCK(chain)
5:          ⟨parent, transactions, proof⟩ ← block
6:          if parent ≠ HASH(previousBlock) ∨
                  not VALIDTRANSACTIONS(transactions) ∨
                  not TEE.VALIDATTESTATION(proof) then
7:              return false
8:          end if
9:          ⟨nonce, l⟩ ← TEE.PROOFDATA(proof)
10:         if nonce ≠ HASH(⟨parent, transactions⟩) then
11:             return false
12:         end if
13:         previousBlock ← block
14:         chain ← WITHOUTEARLIESTBLOCK(chain)
15:     end while
16:     return true
17: end function
```

Proof-of-Luck (Cont.)

- 모든 참가자는 아래와 같은 요소들을 가지고 블록체인 프로토콜을 시작한다.
  - 비어있는 블록체인(currentChain = ε),
  - 보류중인 트랜잭션의 집합(transaction = ε)
  - 초기 라운드블록(roundBlock = null)

- newRound을 통해 state가 초기화된 후, network message를 수신한다.
  - 포함되지 않은 transaction이 수신되면 본인의 transactions에 추가한다.
  - 그 후, peer node에게 브로드캐스팅한다.

  - 새로운 chain이 수신되면, VALID인지, higher LUCK인지 검증한다.
  - 그 후, 새로운 체인으로 변경 후 브로드캐스트한다.

Algorithm 8 Proof of luck blockchain protocol

```
1:  currentChain ← ε
2:  transactions ← ε
3:  roundBlock ← null

4:  function NEWROUND(chain)
5:      roundBlock ← LATESTBLOCK(chain)
6:      POLROUND(roundBlock)
7:      RESETCALLBACK(callback, ROUND_TIME)
8:  end function

9:  on transaction from NETWORK
10:     if transaction ∉ transactions then
11:         transactions ← INSERT(transactions, transaction)
12:         NETWORK.BROADCAST(transaction)
13:     end if
14: end on

15: on chain from NETWORK
16:     if VALID(chain) ∧ LUCK(newChain) >
            LUCK(oldChain) then
17:         currentChain ← chain
18:         if roundBlock = null then
19:             NEWROUND(chain)
20:         else
21:             latestBlock ← LATESTBLOCK(chain)
22:             if latestBlock.parent ≠ roundBlock.Parent then
23:                 NEWROUND(chain)
24:             end if
25:         end if
26:         NETWORK.BROADCAST(chain)
27:     end if
28: end on

29: on callback
30:     newTransactions ← transactions
31:     transactions ← ε
32:     chain ← COMMIT(newTransactions, currentChain)
33:     NETWORK.SENDTOSELF(chain)
34: end on
```

Proof-of-Luck (Cont.)

- 다음의 경우, 브로드캐스팅 전에 참가자는 NewRound를 호출하게 된다.
  - 첫 번째 라운드일 경우 or 새로운 체인과 자신의 최신블록의 parent가 상이할 경우
  - 채굴 라운드를 진행하며 계속해서 더 운이 좋은 체인에 대한 메시지를 수신한다.
  - 만약 더 운이 좋은 체인이 나타나면 해당 체인으로 변경한다.
  - 이 때, parent는 변하지 않지만,
  - 참가자가 network split의 일부분이라면 parent는 상이할 것이고,
  - 이 경우 참가자는 새로운 체인에서 다시 채굴을 시작하게 된다.

- newRound를 호출함으로써 새로운 라운드가 시작되면,
- PoLRound를 통해 PoLMine이 실행될 것이고, 보유중인 callback을 지운 후
- ROUND_TIME이후 시작하도록 새 콜백을 예약한다.
- 이것은 모든 참가자가 ROUND_TIME마다 한 번의 채굴을 한다는 것을 의미한다.
- callback 내에서 보류중인 트랜잭션은 commit을 통해 새로운 체인에 추가되고,
- 새로운 체인은 network.sendToSelf를 통해 다시 전송된다.

Algorithm 8 Proof of luck blockchain protocol

```
1:  currentChain ← ε
2:  transactions ← ε
3:  roundBlock ← null

4:  function NEWROUND(chain)
5:      roundBlock ← LATESTBLOCK(chain)
6:      POLROUND(roundBlock)
7:      RESETCALLBACK(callback, ROUND_TIME)
8:  end function

9:  on transaction from NETWORK
10:     if transaction ∉ transactions then
11:         transactions ← INSERT(transactions, transaction)
12:         NETWORK.BROADCAST(transaction)
13:     end if
14: end on

15: on chain from NETWORK
16:     if VALID(chain) ∧ LUCK(newChain) >
            LUCK(oldChain) then
17:         currentChain ← chain
18:         if roundBlock = null then
19:             NEWROUND(chain)
20:         else
21:             latestBlock ← LATESTBLOCK(chain)
22:             if latestBlock.parent ≠ roundBlock.Parent then
23:                 NEWROUND(chain)
24:             end if
25:         end if
26:         NETWORK.BROADCAST(chain)
27:     end if
28: end on

29: on callback
30:     newTransactions ← transactions
31:     transactions ← ε
32:     chain ← COMMIT(newTransactions, currentChain)
33:     NETWORK.SENDTOSELF(chain)
34: end on
```

- 참가자는 동기화된 clock을 가질 필요가 없지만,
- 프로토콜에 의해 라운드가 동기화되는 경향이 있다.

- PoL은 더 운이 좋은 proof를 release하기 때문에,
- 참가자는 항상 더 운이 좋은 새로운 체인을 받게 된다.

- 만약 참가자가 더 운이 좋지 않은 체인을 얻게 된다면,
- 해당 체인은 브로드캐스트 하지 않는다.

Q & A