ASIC-RESISTANT

CONSENSUS ALGORITHM BASED BLOCKCHAIN

REFERENCES

- Go-ethereum Pink Marble (v1.9.14)
  - https://github.com/ethereum/go-ethereum/tree/6d74d1e5f762e06a6a739a42261886510f842778
- 박상현, and 문수묵. "ASIC 저항 알고리즘에 기반한 블록체인의 릴레이 기술 현황 조사." 한국정보과학회 학술발표논문집 (2019): 806-808.

PROOF-OF-WORK

NORMAL POW-BASED BLOCKCHAIN

- 작업증명(Proof-of-Work, PoW) 기반 블록체인
  - 채굴자(miner)들은 논스(nonce)를 변경하며
  - 유효한 블록 해시를 찾고자 함
- $Hash(header) \leq Difficulty$

NORMAL POW-BASED BLOCKCHAIN

- 유효한 블록 헤더를 찾기까지는 많은 양의 연산이 필요
  - CPU
  - GPU
  - ASIC (Application-Specific Integrated Circuit)

ASIC-RESISTANT

ASIC-RESISTANT

- ASIC 채굴기의 해시 파워가 압도적으로 높음
- ASIC 저항 합의 알고리즘으로
  - CPU 및 GPU 채굴자에게 공정한 기회를 주고자 함

ASIC-RESISTANT

- 이더리움의 ASIC 저항 합의 알고리즘
  - Ethash
  - ProgPoW

ETHASH

ETHASH

- 메모리 I/O 병목을 활용해 ASIC 저항성을 제공
- 유효성 검증을 위해
  - 1GiB 이상의 데이터셋 또는
  - 16MiB 이상의 캐시와 많은 양의 SHA3-512 연산을 요구

MINE

- ./consensus/ethash/sealer.go L164

- // Compute the PoW value of this nonce
  digest, **result** := **hashimotoFull**(dataset.dataset, hash, nonce)
  if **new(big.Int).SetBytes(result).Cmp(target) <= 0** {
  // Correct nonce found, ⋯
  }

HASHIMOTOFULL

- 1GiB 이상의 데이터셋으로 유효성 검증
  - 전체 **인메모리 데이터셋**을 사용

HASHIMOTOFULL

- ./consensus/ethash/algorithm.go L395

- ```
  func hashimotoFull(dataset []uint32, hash []byte, nonce uint64) ([]byte, []byte) {
      lookup := func(index uint32) []uint32 {
          offset := index * hashWords
          return dataset[offset : offset+hashWords]
      }
      return hashimoto(hash, nonce, uint64(len(dataset))*4, lookup)
  }
  ```
  ```

HASHIMOTO

- ./consensus/ethash/algorithm.go L334
- func hashimoto

input

```
(hash []byte, nonce uint64, size uint64, lookup func(index uint32) []uint32)
```

```
([]byte, []byte)
```

HASHIMOTO

- ./consensus/ethash/algorithm.go L334
- func hashimoto

  (hash []byte, nonce uint64, size uint64, lookup func(index uint32) []uint32)

  ([]byte, []byte) output

HASHIMOTO

- rows 계산
  - rows := uint32(size / mixBytes)

HASHIMOTO

- seed, seedHead 계산
  - $seed = keccak512(header||nonce)$
- ```
  seed := make([]byte, 40)
  copy(seed, hash)
  binary.LittleEndian.PutUint64(seed[32:], nonce)
  seed = crypto.Keccak512(seed)
  seedHead := binary.LittleEndian.Uint32(seed)
  ```
  ```

HASHIMOTO

- Mix 계산

- ```
  mix := make([]uint32, mixBytes/4)
  for i := 0; i < len(mix); i++ {
      mix[i] = binary.LittleEndian.Uint32(seed[i%16*4:])
  }
  ```
  ```

HASHIMOTO

- 랜덤 데이터셋과 Mix 결합

- ```
  temp := make([]uint32, len(mix))
  for i := 0; i < loopAccesses; i++ {
      parent := fnv(uint32(i)^seedHead, mix[i%len(mix)]) % rows
      for j := uint32(0); j < mixBytes/hashBytes; j++ {
          copy(temp[j*hashWords:], lookup(2*parent+j))
      }
      fnvHash(mix, temp)
  }
  ```
  ```

FNV

- ./consensus/ethash/algorithm.go L221
- FNV (Fowler-Noll-Vo) 해시 방법론을 가져옴 (변형)
  - 두 파라미터에 대한 빠른 비암호화 해시
- `func fnv(a, b uint32) uint32 { return a*0x01000193 ^ b }`
-
```
func fnvHash(mix []uint32, data []uint32) {
    for i := 0; i < len(mix); i++ {
        mix[i] = mix[i]*0x01000193 ^ data[i]
    }
}
```

HASHIMOTO

- Mix 압축
  - mixBytes / 4 =128 / 4 = 32 길이에서
  - 32 / 4 = 8 로 압축
- ```
  for i := 0; i < len(mix); i += 4 {
      mix[i/4] = fnv(fnv(fnv(mix[i], mix[i+1]), mix[i+2]), mix[i+3])
  }
  mix = mix[:len(mix)/4]
  ```
  ```

HASHIMOTO

- mix([]uint32)를 digest([]byte)에 대입

- ```
  digest := make([]byte, common.HashLength)
  for i, val := range mix {
      binary.LittleEndian.PutUint32(digest[i*4:], val)
  }
  return digest, crypto.Keccak256(append(seed, digest…))
  ```
  ```

HASHIMOTO

- $seed \rightarrow mix \rightarrow mix_{dataset} \rightarrow mix_{compressed} \rightarrow digest \rightarrow result$
- result = crypto.Keccak256(append(seed, digest...))
  - mine에서 result로 사용됨
  - $Hash(header||dataset) \leq Difficulty$

CACHE & DATASET

DATASET

- ./consensus/ethash/ethash.go L284

- ```
  type dataset struct {
      epoch   uint64    // Epoch for which this cache is relevant
      dump    *os.File  // File descriptor of the memory mapped cache
      mmap    mmap.MMap // Memory map itself to unmap before releasing
      dataset []uint32  // The actual cache data content
      once    sync.Once // Ensures the cache is generated only once
      done    uint32    // Atomic flag to determine generation status
  }
  ```
  ```

MAKEDATASET

- ./consensus/ethash/ethash.go L385
- 에폭(epoch) 계산
  - $d.epoch = blockNumber/30000$
- ```
  func MakeDataset(block uint64, dir string) {
      d := dataset{epoch: block / epochLength}
      d.generate(dir, math.MaxInt32, false, false)
  }
  ```
  ```

GENERATE

- ./consensus/ethash/ethash.go L300

- ```
  csize := cacheSize(d.epoch*epochLength + 1)
  dsize := datasetSize(d.epoch*epochLength + 1)
  seed := seedHash(d.epoch*epochLength + 1)
  ```
  ```

CACHE SIZE & DATASET SIZE

- ./consensus/ethash/algorithm.go L53
- Cache size 계산
-
```
func cacheSize(block uint64) uint64 {
    epoch := int(block / epochLength)
    if epoch < maxEpoch {
        return cacheSizes[epoch]
    }
    return calcCacheSize(epoch)
}
```

CACHE SIZE & DATASET SIZE

- cacheSizes는 첫 2048 에폭에 대한 룩업 테이블(lookup table)
  - 2048*30000 = 61440000 블록에 대한 캐시 사이즈 하드코딩
- var cacheSizes = [maxEpoch]uint64{
  16776896, 16907456, 17039296, 17170112, 17301056, 17432512, 17563072,
  17693888, 17824192, 17955904, 18087488, 18218176, 18349504, 18481088,
  18611392, 18742336, 18874304, 19004224, 19135936, 19267264, 19398208,
  ...
  }

CACHE SIZE & DATASET SIZE

- 61,440,000 블록을 넘어가면
  - calcCacheSize 호출
  - 에폭에 대한 캐시 사이즈 계산
- size / hashBytes 가 소수가 아닐 경우 소수가 될 수 있도록 조정
  - 2^{64} 이하에서는 항상 정확하게 판단하는 **.ProbablyPrime(1)** 사용
  - 사이클릭한 특성(cyclic behavior)을 피하고자 함

CACHE SIZE & DATASET SIZE

- ./consensus/ethash/algorithm.go L74
- Dataset size 계산
- ```
  func datasetSize(block uint64) uint64 {
      epoch := int(block / epochLength)
      if epoch < maxEpoch {
          return datasetSizes[epoch]
      }
      return calcDatasetSize(epoch)
  }
  ```
  ```

CACHE SIZE & DATASET SIZE

- datasetSizes는 첫 2048 에폭에 대한 룩업 테이블(lookup table)
  - 2048*30000 = 61440000 블록에 대한 데이터셋 사이즈 하드코딩
-
```
var datasetSizes = [maxEpoch]uint64{
    1073739904, 1082130304, 1090514816, 1098906752, 1107293056,
    1115684224, 1124070016, 1132461952, 1140849536, 1149232768,
    1157627776, 1166013824, 1174404736, 1182786944, 1191180416,
    ...
}
```

CACHE SIZE & DATASET SIZE

- 61,440,000 블록을 넘어가면
  - calcDatasetSize 호출
  - 에폭에 대한 데이터셋 사이즈 계산
- size / mixBytes 가 소수가 아닐 경우 소수가 될 수 있도록 조정
  - 2^{64} 이하에서는 항상 정확하게 판단하는 **.ProbablyPrime(1)** 사용
  - 사이클릭한 특성(cyclic behavior)을 피하고자 함

SEEDHASH

- ./consensus/ethash/algorithm.go L121
- 검증 캐시와 마이닝 데이터셋 생성에 사용되는 seed 계산
  - epochLength인 30000 블록마다 seed가 바뀜
-
```
func seedHash(block uint64) []byte {
    seed := make([]byte, 32)
    if block < epochLength { return seed }
    keccak256 := makeHasher(sha3.NewLegacyKeccak256())
    for i := 0; i < int(block/epochLength); i++ { keccak256(seed, seed) }
    return seed
}
```

GENERATE

- ./consensus/ethash/ethash.go L300
- 시드로부터 캐시 생성
- 캐시로부터 데이터셋 생성
- cache := make([]uint32, csize/4)
  generateCache(cache, d.epoch, seed)
  d.dataset = make([]uint32, dsize/4)
  generateDataset(d.dataset, d.epoch, cache)

GENERATECACHE

- ./consensus/ethash/algorithm.go L139
- 32MB 메모리 채우기
- Sergio Demian Lerner의 **RandMemoHash** 알고리즘
  - Strict Memory Hard Hashing Functions (2014)
- 524288개 64바이트 아웃풋

GENERATECACHE

- Ref. Lerner, Sergio Demian. "STRICT MEMORY HARD HASHING FUNCTIONS”

- $\mathbf{RandMemoHash}(s, R, N)$

(1) Set $M[0] := s$
(2) For $i := 1$ to $N - 1$ do set $M[i] := H(M[i - 1])$
(3) For $r := 1$ to R do
  (a) For $b := 0$ to $N - 1$ do
    (i) $p := (b - 1 + N) \bmod N$
    (ii) $q :=$AsInteger$(M[p]) \bmod (N - 1)$
    (iii) $j := (b + q) \bmod N$
    (iv) $M[b] :=$H$(M[p] \mid\mid M[j])$

GENERATECACHE

- rows 계산
- size := uint64(len(cache))
  rows := int(size) / hashBytes

GENERATECACHE

- 초기 데이터셋 생성

- keccak512(cache, seed)
```
for offset := uint64(hashBytes); offset < size; offset += hashBytes {
    keccak512(cache[offset:], cache[offset-hashBytes:offset])
    atomic.AddUint32(&progress, 1)
}
```

- $\mathbf{RandMemoHash}(s, R, N)$

  (1) Set $M[0] := s$
  (2) For $i := 1$ to $N - 1$ do set $M[i] := H(M[i - 1])$

GENERATECACHE

- 저-라운드 버전의 randmemohash
-
```
for i := 0; i < cacheRounds; i++ {
    for j := 0; j < rows; j++ {
        var (
            srcOff = ((j - 1 + rows) % rows) * hashBytes
            dstOff = j * hashBytes
            xorOff = (binary.LittleEndian.Uint32(cache[dstOff:]) % uint32(rows)) * hashBytes
        )
        bitutil.XORBytes(temp, cache[srcOff:srcOff+hashBytes], cache[xorOff:xorOff+hashBytes])
        keccak512(cache[dstOff:], temp)
        atomic.AddUint32(&progress, 1)
    }
}
```

GENERATECACHE

- for i := 0; i < cacheRounds; i++ {
  for j := 0; j < rows; j++ { ⋯ }}

- (3) For $r := 1$ to R do
  (a) For $b := 0$ to $N - 1$ do
  (i) $p := (b - 1 + N) \bmod N$
  (ii) $q :=$AsInteger$(M[p]) \bmod (N - 1)$
  (iii) $j := (b + q) \bmod N$
  (iv) $M[b] :=$H$(M[p] \mid\mid M[j])$

- $R = 3$
  $N = rows = csize/64$

GENERATEDATASET

- ./consensus/ethash/algorithm.go L267
- 캐시로부터 데이터셋 생성
- 멀티쓰레드
-
```
for index := first; index < limit; index++ {
    item := generateDatasetItem(cache, index, keccak512)
    ...
    copy(dataset[index*hashBytes:], item)
    ...
}
```

GENERATEDATASETITEM

- ./consensus/ethash/algorithm.go L234
- 256개의 수도랜덤하게 선택된 캐시 노드를 해싱해
- 하나의 데이터셋 노드를 생성하는 과정을
- 반복

GENERATEDATASETITEM

- rows 계산
- rows := uint32(len(cache) / hashWords)

GENERATEDATASETITEM

- mix 초기화
- mix := make([]byte, hashBytes)
- binary.LittleEndian.PutUint32(mix, **cache**[(**index**%rows)*hashWords]^index)
  for i := 1; i < hashWords; i++ {
  binary.LittleEndian.PutUint32(
  mix[i*4:], **cache**[(**index**%rows)*hashWords+**uint32(i)**]
)
  }
- keccak512(mix, mix)

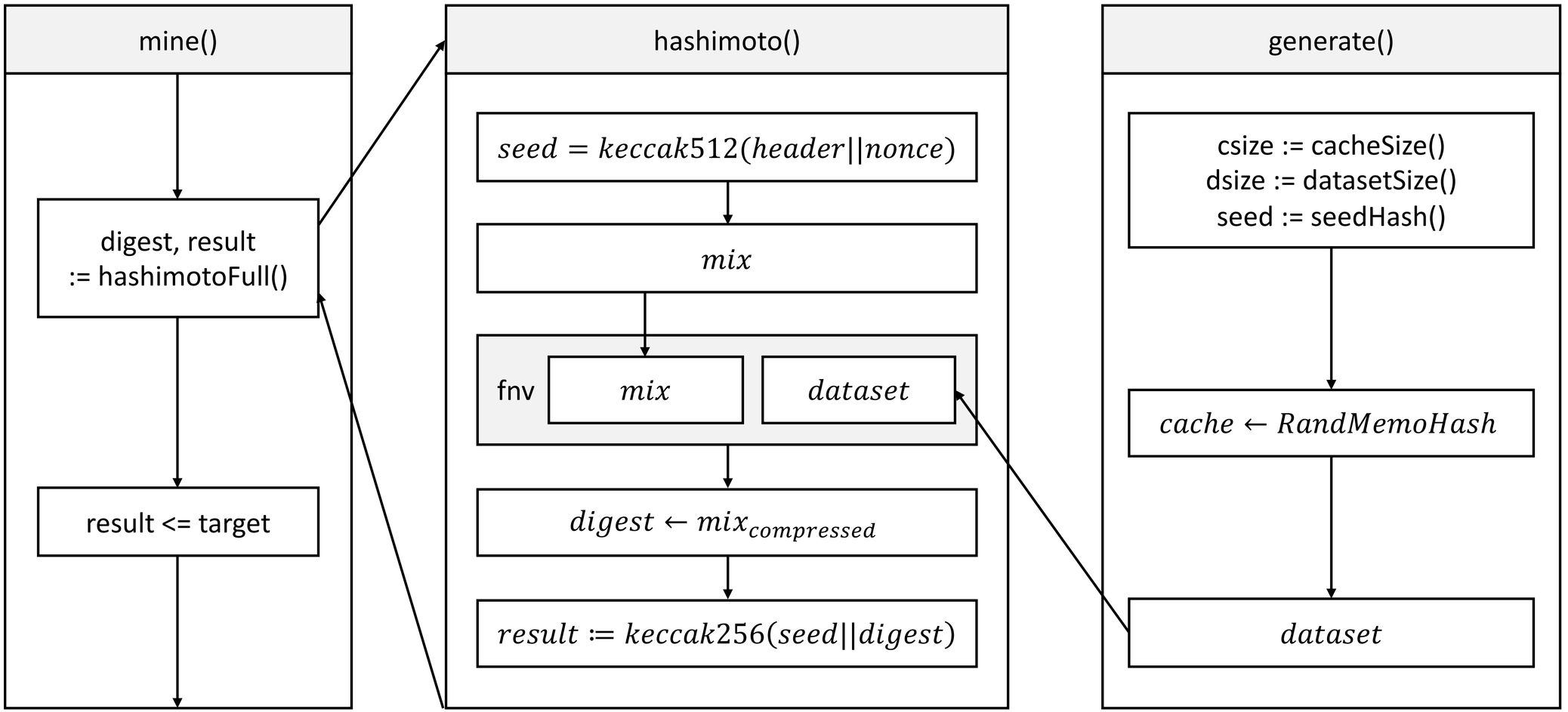
GENERATEDATASETITEM

- 바이트 배열인 mix를 uint32 배열인 intMix로 변경

- ```
  intMix := make([]uint32, hashWords)
  for i := 0; i < len(intMix); i++ {
      intMix[i] = binary.LittleEndian.Uint32(mix[i*4:])
  }
  ```
  ```

GENERATEDATASETITEM

- 많은 양(256)의 랜덤 캐시와 FNV
-
```
for i := uint32(0); i < datasetParents; i++ {
    parent := fnv(index^i, intMix[i%16]) % rows
    fnvHash(intMix, cache[parent*hashWords:])
}
```

GENERATEDATASETITEM

- intMix —> mix
- 해시값을 리턴
- ```
  for i, val := range intMix {
      binary.LittleEndian.PutUint32(mix[i*4:], val)
  }
  keccak512(mix, mix)
  return mix
  ```
  ```

SUMMARY

ETHASH


mine()
digest, result
:= hashimotoFull()
result <= target
hashimoto()
$seed = keccak512(header||nonce)$
mix
fnv
mix
$dataset$
$digest \leftarrow mix_{compressed}$
$result := keccak256(seed||digest)$
generate()
csize := cacheSize()
dsize := datasetSize()
seed := seedHash()
$cache \leftarrow RandMemoHash$
$dataset$

PROGPOW

PROGPOW

- Ethash 기반 ASIC은 설계하기 어려움
  - 불가능하지는 않음
- 실제로 이더리움 해시레이트의 40%가 채굴기로부터 나옴
  - Ref. Kristy-Leigh Minehan. Fingerprinting of the Bitmain Antminer E3 via Statistical Nonce Analysis [Online]. Available: https://twitter.com/OhGodAGirl/status/1235824954291015680 (downloaded 2020, Mar. 30)

PROGPOW

- 프로그래매틱 작업증명 (Programmatic Proof-of-Work, ProgPoW)
  - Ethash의 ASIC 저항성을 확장
  - 현재의 ASIC이 제공하지 못하는 GPU의 범용적인 기능을 모두 활용
- 근본적인 해결책은 아님
- 성급한 도입은 블록체인 네트워크의 분열을 야기할 수 있음

ASIC-RESISTANT

CONSENSUS ALGORITHM BASED BLOCKCHAIN