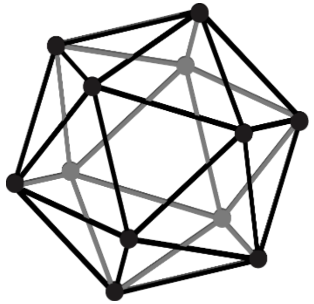
Paper Review: New Consensus Protocol

Proof of Luck: an Efficient Blockchain Consensus Protocol

https://youtu.be/m5Ke-R8d7Cc



Proof of **luck**: An efficient blockchain consensus protocol

M Milutinovic, W He, H Wu, M Kanwal - … of the 1st Workshop on System …, 2016 - dl.acm.org

In the paper, we present designs for multiple blockchain consensus primitives and a novel blockchain system, all based on the use of trusted execution environments (TEEs), such as Intel SGX-enabled CPUs. First, we show how using TEEs for existing **proof** of work schemes …

Bitcoin

- Blockchain System
- Cryptocurrency
- Distributed Ledger
  - Mutual Distrust Participants
  - Require Consensus Algorithm

  → Proof of Work

Proof of Work (PoW)

Bitcoin's electricity consumption surpasses Singapore and Portugal

February 19, 2018 | Sustainability | 3 Comments

According the Bitcoin Energy Consumption Index the whole Bitcoin network now consumes more than 50 terawatt-hours (TWh) of electricity per year. This marks a new major milestone for the increasing energy-hungry network, which has now surpassed countries like Singapore (49.5 TWh per year) and Portugal (49.8 TWh per year) in terms of electricity consumption.

https://digiconomist.net/bitcoin-electricity-consumption-surpasses-singapore-portugal

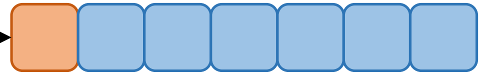
- Consensus Algorithms
- Cons
  - High Cost (time, energy)

Uncle, Dividing Block

Proof of Stake, Proof of Burn

- Bitcoin's PoW
  - Block time : 10 min
  - Confirm : after 6 blocks

Decrease the Number of Forks



New Consensus Model Using TEEs

- Trusted Execution Environments (TEE)
  - Intel SGX
- Pros
  - Enforce correct processing
  - Limit the effect of Sybils → Sybil Attack : Control multiple node

Improving Blockchain Through Technology Innovation

Intel® Software Gua
transactions, and he
cryptographic hash
like Intel® Advanced
into Intel® Xeon® Sc

Learn more about

Intel's Commitment to Open Source

In 2016 Intel contributed Sawtooth to Hyperledger*, the open source consortium hosted by The Linux Foundation. Hyperledger Sawtooth* is an enterprise-grade blockchain platform for building distributed ledger applications and networks. The design philosophy targets keeping ledgers distributed and making smart contracts safer, particularly for enterprise use.

Learn more about Hyperledger Sawtooth* software

https://www.intel.com/content/www/us/en/security/blockchain-overview.html

Consensus

- Quick and Deterministic transaction confirmations.
- Energy efficient protocol.
- Resistant to custom hardware. (ASIC-resistant)

<table>
<tr><th>GPU</th><th>Mhash/s</th><th>ASICs</th><th>Mhash/s</th></tr>
<tr><td>Titan X</td><td>1,980</td><td>AntMiner S7</td><td>4,730,000</td></tr>
<tr><td>GTX 980 Ti</td><td>4,500</td><td rowspan="2">Avalon 6</td><td rowspan="2">3,500,000</td></tr>
<tr><td>GTX 960 Gaming 2GOC</td><td>1,173</td></tr>
<tr><td>GTX 1080</td><td>2,048</td><td rowspan="2">SP20 Jackson</td><td rowspan="2">1,500,000</td></tr>
<tr><td>GTX 1060</td><td>1,800</td></tr>
</table>

http://wiki.hash.kr/index.php/%EC%97%90%EC%9D%B4%EC%8B%9D

Proof of Work (inside TEE)

Algorithm 1 TEE-enabled proof of work

```
function PoW(nonce, difficulty)
    result ← ORIGINALPoW(nonce, diffic
    assert ORIGINALPoWSUCCESS(result)
    return TEE.ATTESTATION(⟨nonce, difficulty⟩, null)
end funct
```

sgx_create_monotonic_counter

`sgx_create_monotonic_counter` creates a monotonic counter with default owner policy and default user attribute mask.

- Code unmodified

ide TEE)

EE)

COUNTER()

```
    assert counter = READMONOTONICCOUNTER()
    return TEE.ATTESTATION(⟨nonce, duration⟩, null)
end function
```

Enroll in Intel SGX Attestation Service

One of the key decisions when subscribing to the Intel SGX attestation service is the mode chosen for the EPID signature, Random Base Mode or Name Base Mode. Additional background on EPID signature modes as well as provisioning and attestation services, please see this white paper.

Linkable Quotes (Name Base Mode): A name is picked for the base to be used for a signature, making signatures linkable. Verifying two signatures enables you to tell whether they were generated from the same or different signers. Name Base Mode is preferred to protect against compromise.

- Sybil-resistant

Proof of Ownership (inside TEE)

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

```
function PoO(nonce)
    return TEE.ATTESTATION(⟨nonce⟩, nonce)
end function
```

- Benefits
    - Sybil-resistant

Proof of Luck (inside TEE)

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

```
counter ← INCREMENTMONOTONICCOUNTER()
roundBlock ← null
roundTime ← null

function POLROUND(block)        Currently known latest block
    roundBlock ← block
    roundTime ← TEE.GETTRUSTEDTIME()
end function                    NewBlock's            After ROUND_TIME

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

    roundBlock ← null
    roundTime ← null
    l ← GETRANDOM() [0, 1), from uniform distribution
    SLEEP(f(l)) Short delay for luckier, Long delay for unluckier

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

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

Algorithm 5 Extending a blockchain with a new block

```
function COMMIT(newTransactions, chain)
    previousBlock ← LATESTBLOCK(chain)
    parent ← HASH(previousBlock)
    header ← ⟨parent, newTransactions⟩
    proof ← POLMINE(header, previousBlock)
    newBlock ← ⟨parent, newTransactions, proof⟩
    return APPEND(chain, newBlock)
end function
```

Proof of Luck (inside TEE)

Algorithm 6 Computing a luck of a valid blockchain

```
function LUCK(chain)
    luck ← 0
    for block in chain do
        luck ← luck + TEE.PROOFDATA(block.proof).l
    end for
    return luck
end function
```

Algorithm 7 Validating a blockchain

```
function VALID(chain)
    previousBlock ← null
    while chain ≠ ε do
        block ← EARLIESTBLOCK(chain)
        ⟨parent, transactions, proof⟩ ← block
        if parent ≠ HASH(previousBlock) ∨
                not VALIDTRANSACTIONS(transactions) ∨
                not TEE.VALIDATTESTATION(proof) then
            return false
        end if
        ⟨nonce, l⟩ ← TEE.PROOFDATA(proof)
        if nonce ≠ HASH(⟨parent, transactions⟩) then
            return false
        end if
        previousBlock ← block
        chain ← WITHOUTEARLIESTBLOCK(chain)
    end while
    return true
end function
```

Proof of Luck Blockchain (inside TEE)

Algorithm 8 Proof of luck blockchain protocol

```
currentChain ← ε
transactions ← ε
roundBlock ← null

function NEWROUND(chain)
    roundBlock ← LATESTBLOCK(chain)
    POLROUND(roundBlock)
    RESETCALLBACK(callback, ROUND_TIME)
end function

on transaction from NETWORK
    if transaction ∉ transactions then
        transactions ← INSERT(transactions, transaction)
        NETWORK.BROADCAST(transaction)
    end if
end on

on chain from NETWORK
    if VALID(chain) ∧ LUCK(newChain) >
            LUCK(oldChain) then
        currentChain ← chain
        if roundBlock = null then
            NEWROUND(chain)
        else
            latestBlock ← LATESTBLOCK(chain)
            if latestBlock.parent ≠ roundBlock.Parent then
                NEWROUND(chain)
            end if
        end if
        NETWORK.BROADCAST(chain)
    end if
end on

on callback
    newTransactions ← transactions
    transactions ← ε
    chain ← COMMIT(newTransactions, currentChain)
    NETWORK.SENDTOSELF(chain)
end on
```

Thanks

vexyoung@gmail.com