Blockchain & Smart Contract Attack Vectors

Jorden Seet

Who am I?

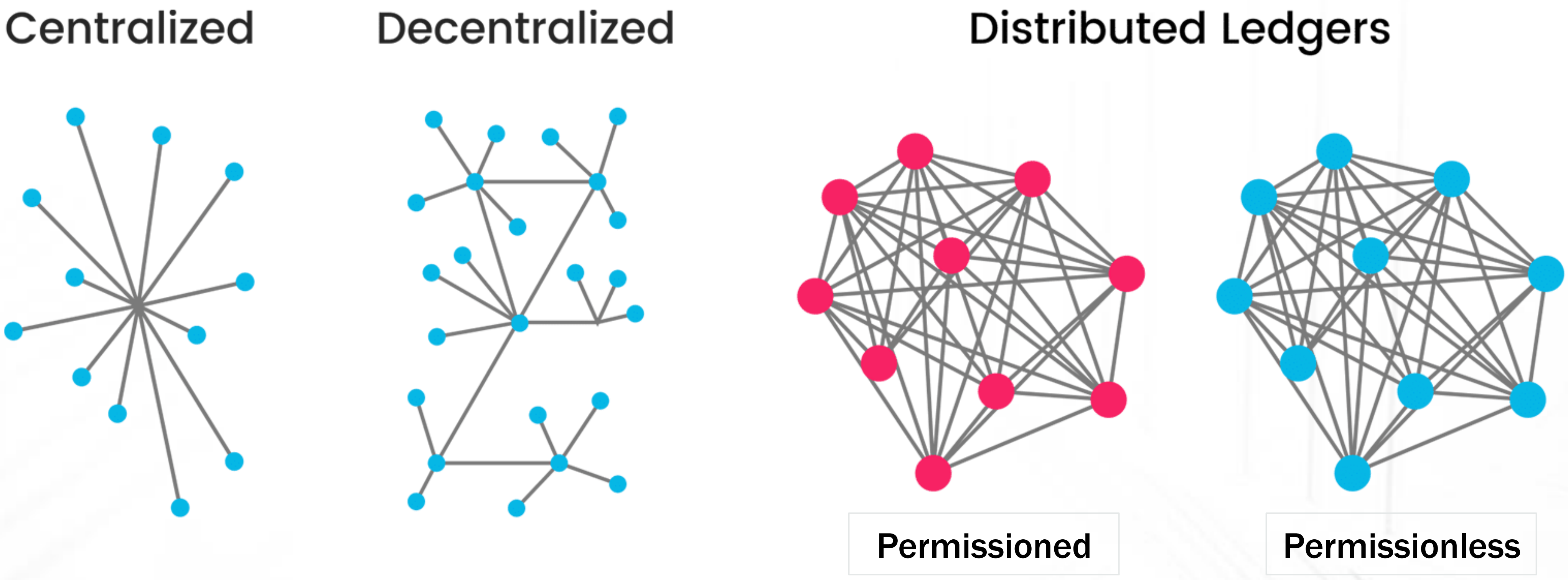
- Jorden Seet
- SMU, BSc Information Systems & Analytics double major
- Blockchain Engineer at BMW Group
- Blockchain Teaching Assistant for SMU
- Passionate about Cybersecurity, from Pentesting to Cryptography to Network Topology
- Recently completed an internship at Cyber Security Agency of Singapore (CSA)
  - Penetration Testing Department

Why am I here?

- Blockchain is a buzzword
  - Foreign?
  - Unhackable?
- It’s not that different!
- Compilation of various attack vectors
  - Usually, presentations focus on a single scope
- Take home something new
  - Smart Contract Attack Vectors
  - System Attack Vectors
  - Topology Attack Vectors

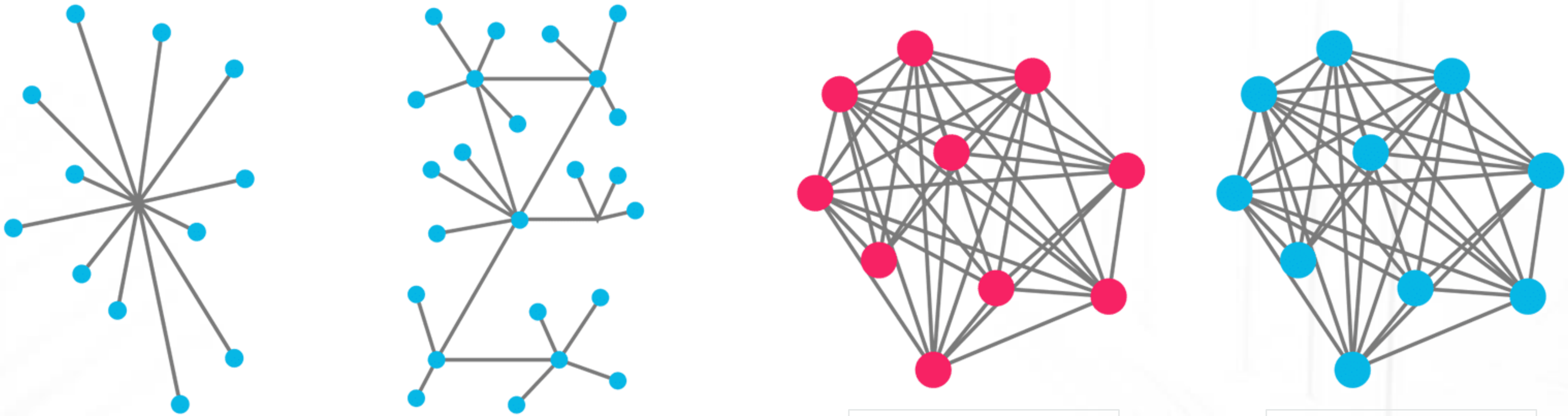
What is a Blockchain?

Blockchain is a special kind of Database



Blockchain is a Distributed Ledger

Centralized | Decentralized | Distributed Ledgers



Permissioned | Permissionless

Benefits of Blockchain

- Data Integrity
  - Consensus rejects nodes with inconsistent data
  - Requires “hacking” into 51% or 33% of nodes to carry attacks
- High Availability
  - As long as one node is up, information is available
  - Hard to DDoS network when there is no single source
  - Distribution of bandwidth can also mitigate DDoS
- High Auditability
  - Information in blockchain is transparent to all nodes in network
  - All information is linked to each other

Smart Contracts

What are Smart Contracts?

- Cryptographic computer protocol that is designed to
  - Facilitate
  - Monitor
  - Verify
  - Enforce
- The performance of an agreement in an <u>immutable</u> and <u>exact</u> manner.
- Without third parties like
  - Lawyers
  - Auditors
  - Witnesses
  - Banks
  - Governments

Why do Smart Contracts need Blockchain?

- Smart Contracts require witnesses
  - Honest nodes operating (anonymously) in a distributed network
- Nodes may not necessarily be honest
- Smart Contracts need a Byzantine Fault Tolerant environment
  - Consensus must be reached amidst failure of nodes
  - Ensured (to a higher degree) through the Blockchain

Smart Contract Attack Vectors

Integer Underflow/Overflow

- When one puts in a value that exceeds the limit, the value becomes something else.
- Uints have a max value of $2^{256} - 1$, in Solidity it wraps after max value to 0

```
CBlock(hash=0000000000790ab3, ver=1, hashPrevBlock=0000000000606865, hashMerkleRoot=618eba,
nTime=1281891957, nBits=1c00800e, nNonce=28192719, vtx=2)
  CTransaction(hash=012cd8, ver=1, vin.size=1, vout.size=1, nLockTime=0)
    CTxIn(COutPoint(000000, -1), coinbase 040e80001c028f00)
    CTxOut(nValue=50.51000000, scriptPubKey=0x4F4BA55D1580F8C3A8A2C7)
  CTransaction(hash=1d5e51, ver=1, vin.size=1, vout.size=2, nLockTime=0)
    CTxIn(COutPoint(237fe8, 0), scriptSig=0xA87C02384E1F184B79C6AC)
    CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HASH160 0xB7A7)
    CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HASH160 0x1512)
  vMerkleTree: 012cd8 1d5e51 618eba

Block hash: 0000000000790ab3f22ec756ad43b6ab569abf0bddeb97c67a6f7b1470a7ec1c
Transaction hash: 1d5e512a9723cbef373b970eb52f1e9598ad67e7408077a82fdac194b65333c9
```

92.2 Billion BTC each!

Timestamp Dependence

- Dangerous to attempt pseudo-random number generation via block timestamps
  - Deterministic, attackers can determine the “random” variable

```
contract CoinFlip {
  uint256 public consecutiveWins;
  uint256 lastHash;
  uint256 FACTOR = 57896044618658097711785492504343953926634992332820282019728792003956564819968;

  function CoinFlip() public {
    consecutiveWins = 0;
  }

  function flip(bool _guess) public returns (bool) {
    uint256 blockValue = uint256(block.blockhash(block.number-1));

    if (lastHash == blockValue) {
      revert();
    }

    lastHash = blockValue;
    uint256 coinFlip = blockValue / FACTOR;
    bool side = coinFlip == 1 ? true : false;
```

Denial of Service – Smart Contract level

- Smart contracts can indicate if they are "payable"
  - Non-payable functions/contracts cannot receive cryptocurrency

```
contract Auction {
    address highestBidder;
    uint highestBid;

    function bid() {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

Reentrancy Attack

```
function payout() public payable {

  checkPermissions(msg.sender);

 if (game.originator.status == STATUS_TIE && game.taker.status == STATUS_TIE) {
   game.originator.addr.transfer(game.betAmount);
   game.taker.addr.transfer(game.betAmount);
 } else {
    if (game.originator.status == STATUS_WINNER) {
      game.originator.addr.transfer(game.betAmount*2);
    } else if (game.taker.status == STATUS_WINNER) {
      game.taker.addr.transfer(game.betAmount*2);
    } else {
      game.originator.addr.transfer(game.betAmount);
      game.taker.addr.transfer(game.betAmount);
    }
 }

 resetGame();
 getBetOutcome();
}
```

When a payment is made, it calls the fallback function of the payee contract

The fallback function can call the payment function again recursively

Phishing using tx.origin

```
interface Wallet {
  function transferTo(address to, uint amount);
}
contract Exploit {
    address owner;
    constructor() public {
        owner = msg.sender;                                              {
    }
    function getOwner() public returns (address) {
        return owner;
    }
    function() payable public {
        Wallet(msg.sender).transferTo(owner, msg.sender.balance);
    }
}
```

Parity Wallet attacks

- Bad coding left them vulnerable to attacks on two occasions

```
contract WalletLibrary is WalletEvents {
  function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized {
    initDaylimit(_daylimit);
    initMultiowned(_owners, _required);
  }
    function execute(address _to, uint _value, bytes _data) external onlyowner returns (bytes32 o_hash) {
        //sends money to the address
  }
}
contract Wallet is WalletEvents {

  function Wallet(address[] _owners, uint _required, uint _daylimit) {
  function() payable {
    // just being sent some cash?
    if (msg.value > 0)
      Deposit(msg.sender, msg.value);
    else if (msg.data.length > 0)
      _walletLibrary.delegatecall(msg.data);
  }
}
```

ialized public{

Short Address Attack

- The fault here lies in the way EVM handles underflows, it pads it with 0s

00000000000000000000000062bec9abe373123b9b635b75608f94eb86441630
0000000000000000000000000000000000000000000000000000000000000002

```
contract MyToken {
    mapping (address => uint) balances;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    function MyToken() {
        balances[tx.origin] = 10000;
    }

    function sendCoin(address to, uint amount) returns(bool sufficient) {
        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[to] += amount;
        Transfer(msg.sender, to, amount);
        return true;
    }

    function getBalance(address addr) constant returns(uint) {
        return balances[addr];
    }
}
```

0x62bec9abe373123b9b635b75608f94eb86441630

2

2<<4 = 32 coins
= 30 new coins

Honeypots

```
contract MultiplicatorX3 {
    address public Owner = msg.sender;

    function() public payable{}

    function withdraw() payable public{
        require(msg.sender == Owner);
        Owner.transfer(this.balance);
    }

    function Command(address adr,bytes data) payable public{
        require(msg.sender == Owner);
        adr.call.value(msg.value)(data);
    }

    function multiplicate(address adr) public payable{
        if(msg.value>=this.balance){
            adr.transfer(this.balance+msg.value);
        }
    }
}
```

this.balance is updated before the multiplicate method is called

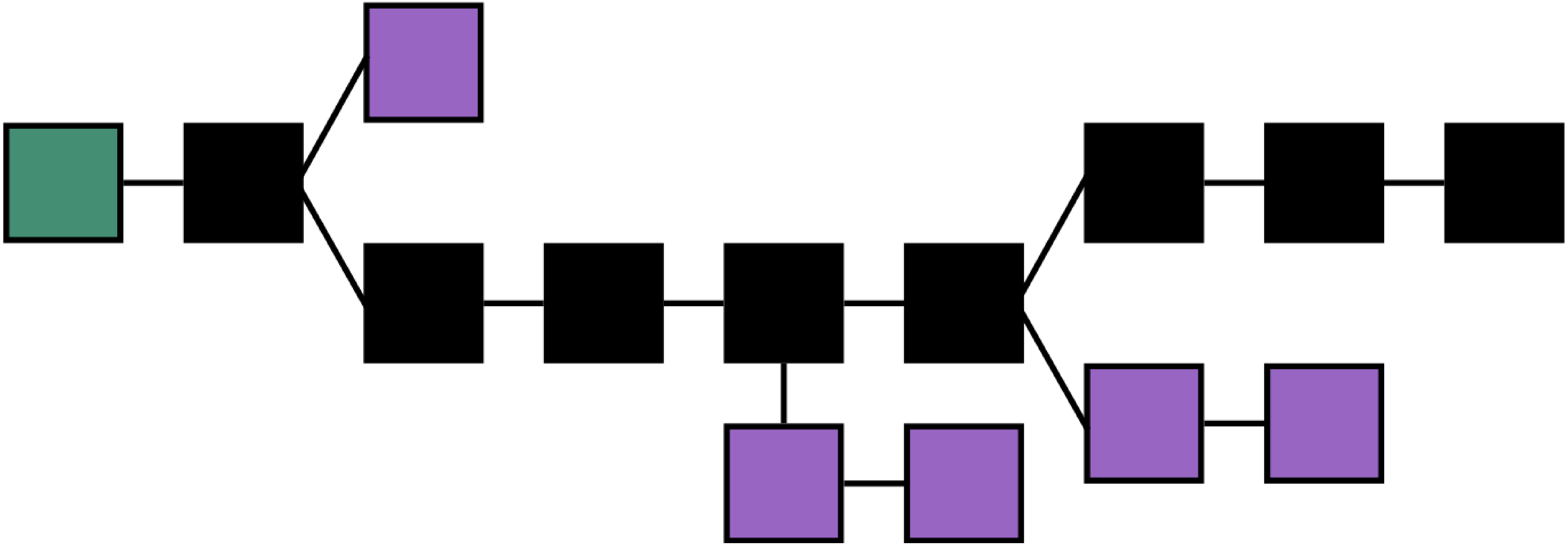
Hence, (msg.value>=this.balance) will always be false and transfer will not run (unless initial balance is 0)

Blockchain Attack Vectors

System Attack Vectors

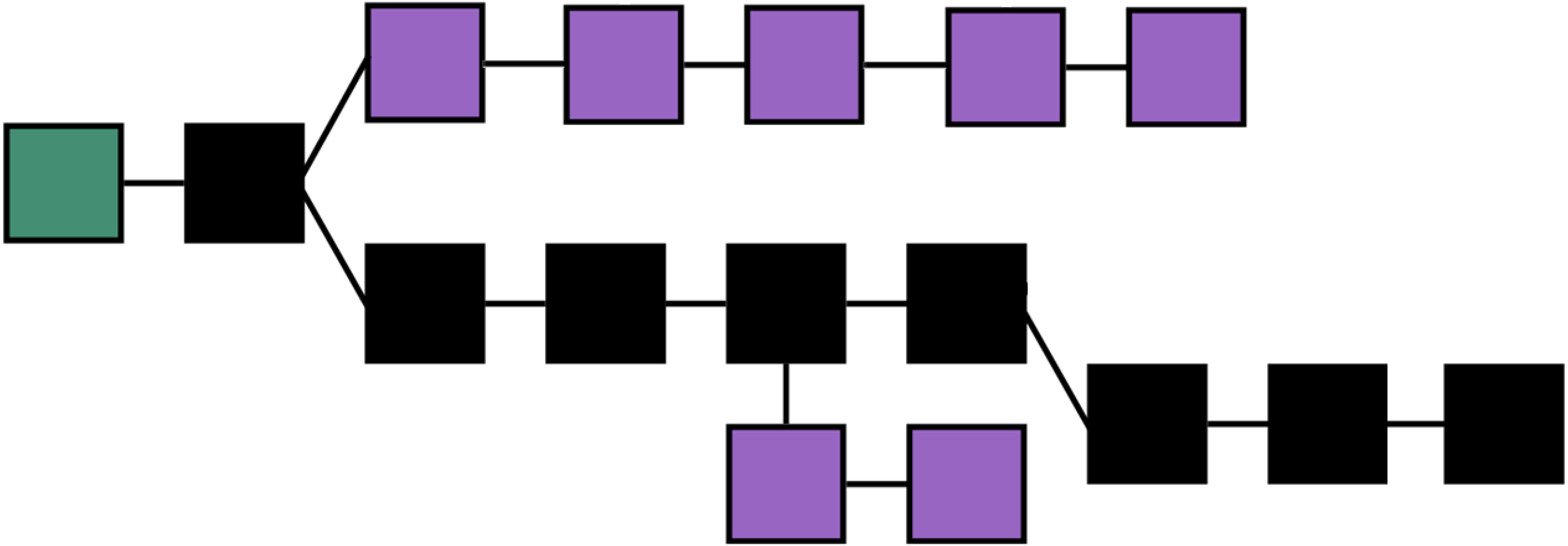
51% Attack (Proof of Work)

- Blocks are validated through mining, which is dependent on hashing power



Long Range Attack (Proof of Stake)

- Blocks are validated through validators, which is dependent on wealth (amount staked)



Transaction (sig) Malleability

- Broadcasting the same transaction signature with a modified transaction hash
  - Sometimes, transaction signatures do not encompass all data in the transaction
  - Requires broadcast before confirmation of transaction into a valid block
- Possible to malform transaction hashes by
  - Modifying minor details (whitespaces or paddings)
  - Using complementary signatures of certain cryptographic signature schemes (ECDSA)
  - Major transaction details like recipients and value remains unchanged
- This transaction, if confirmed over the other original, will trick the sender to think the transaction failed when it actually succeeded
  - Senders query using their transaction signatures
  - Which is invalidated due to the modified, fraudulent transaction
- Can lead to double-spending

Denial of Service – System level

- EOS RAM Hijack
- EOS uses RAM to execute and store state of smart contracts
- Smart contracts can notify others about specific events, such as token transfer

```
void apply_context::update_db_usage( const account_name& payer, int64_t delta ) {
   if( delta > 0 ) {
      if( !(privileged || payer == account_name(receiver)) ) {
         require_authorization( payer );
      }
   }
   trx_context.add_ram_usage(payer, delta);
}

void apply_context::require_authorization( const account_name& account ) {
   for( uint32_t i=0; i < act.authorization.size(); i++ ) {
     if( act.authorization[i].actor == account ) {
        used_authorizations[i] = true;
        return;
     }
   }
   EOS_ASSERT( false, missing_auth_exception, "missing authority of ${account}", ("account",account)
}
```

If recipient is a contract, does not check if action is authorized by action handler → RAM consumed

Can potentially fill action handler's RAM with garbage notifications, thus using up all bandwidth to process contracts

BFT Consensus algorithms forking

- In BFT type algorithms, Blocks require 66% consensus on transaction validity
- NEO Blockchain
  - Initially used a 2-phase protocol for efficiency and lower complexity
  - Missed the “commit” phase of traditional BFT algos
- Uniformity of message “Proposed blocks” is not agreed upon
  - Led to different blocks formed on the blockchain → Fork
- Caused ensuing blocks to record different previous hashes
  - Consensus cannot be agreed upon and stalled

TEE-based consensus

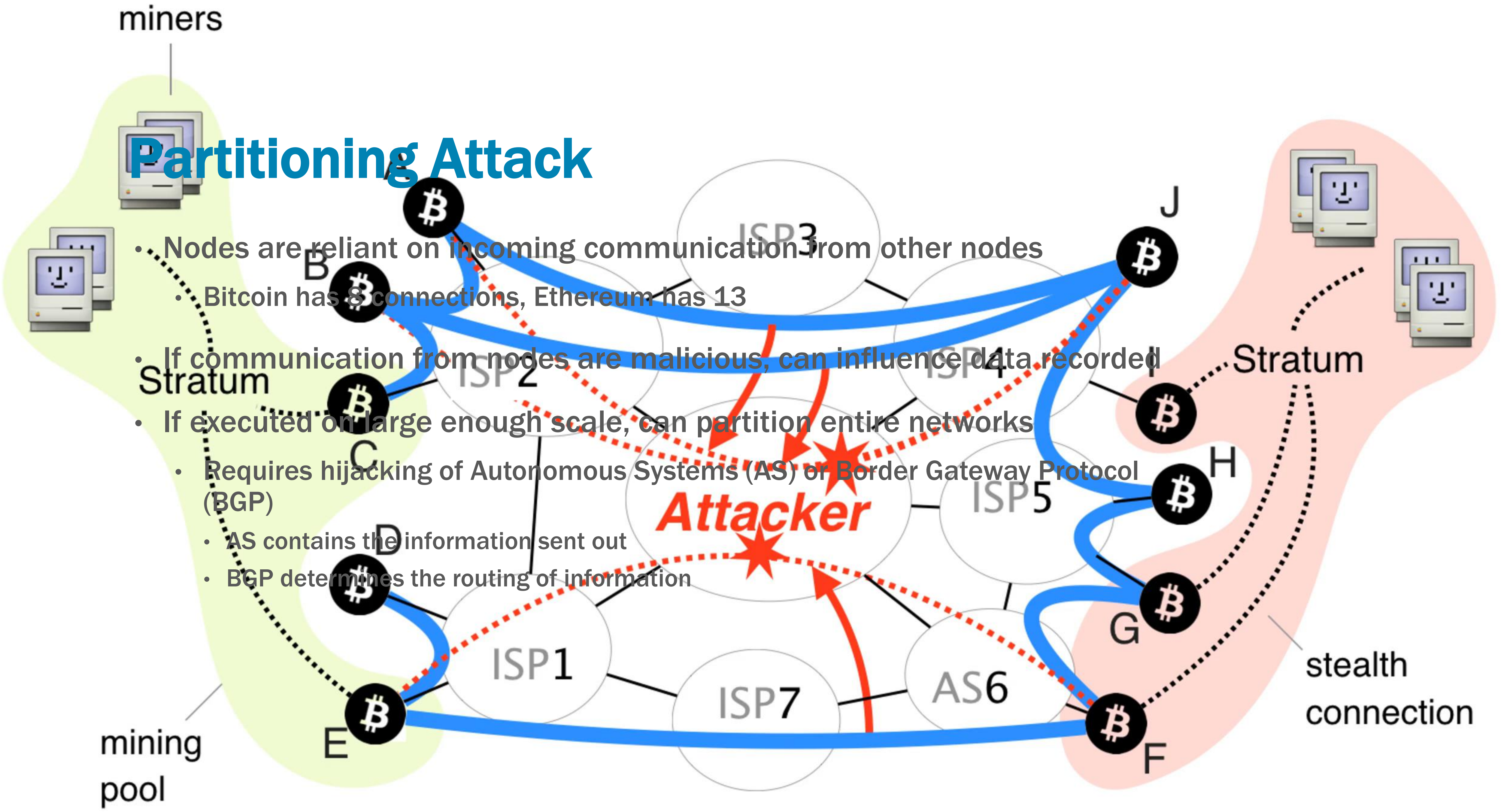
- Proof of Elapsed Time (Intel's Hyperledger Sawtooth)
- Based on Intel Software Guard eXtensions (Intel SGX)
- Utilises Trusted Execution Environments to generate randomness
  - Data is protected from even malicious or hacked kernels
- Speculative Execution
  - Microarchitecturally, processor might speculatively guess values from memory
  - Faulty guesses disturbs other parts of the processor like the cache contents
  - Speculative Execution detects and measures such disturbances to infer in-memory values
- Meltdown, Spectre, Foreshadow

Blockchain Attack Vectors

Topology Attack Vectors

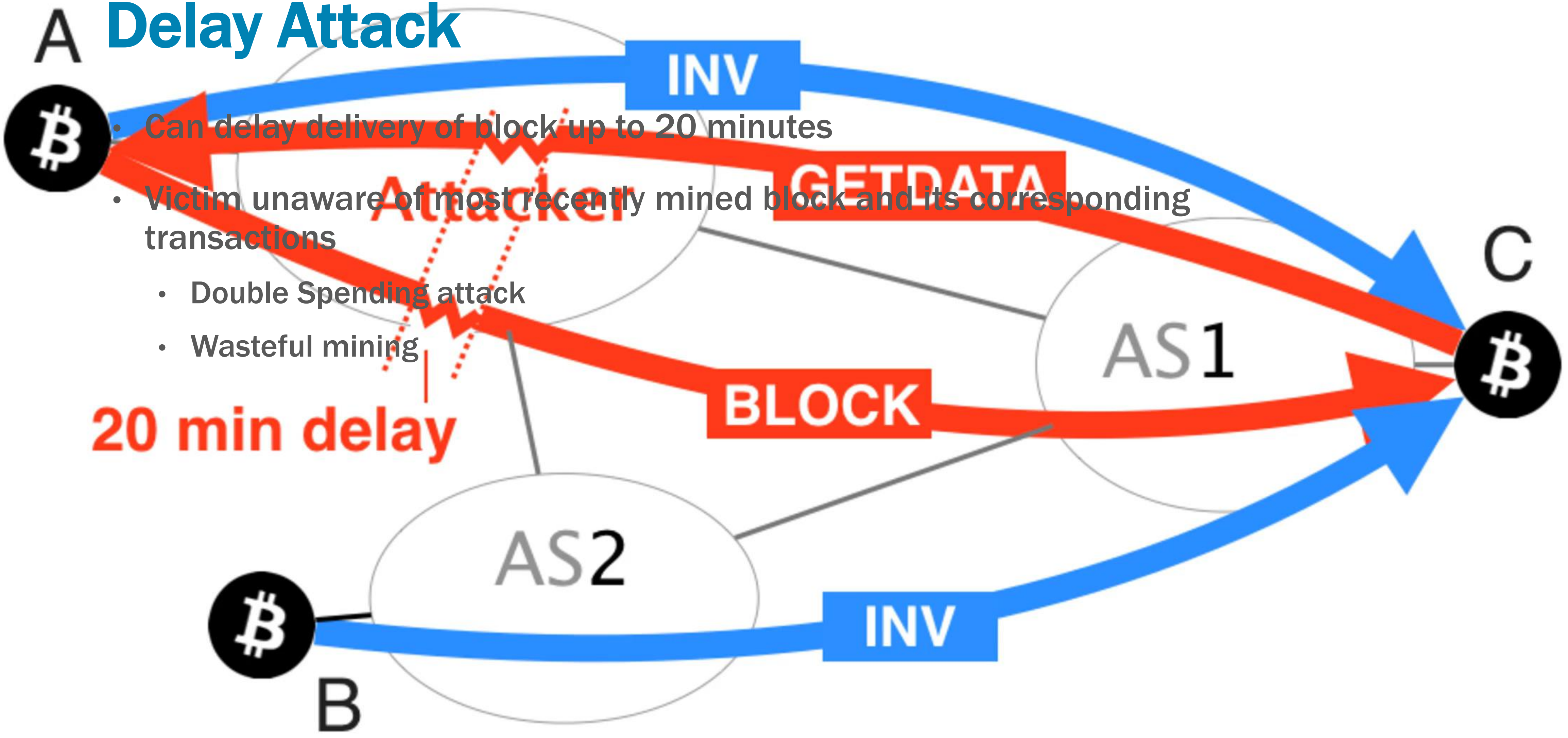
Partitioning Attack

- Nodes are reliant on incoming communication from other nodes
  - Bitcoin has 8 connections, Ethereum has 13
- If communication from nodes are malicious, can influence data recorded
- If executed on large enough scale, can partition entire networks
  - Requires hijacking of Autonomous Systems (AS) or Border Gateway Protocol (BGP)
    - AS contains the information sent out
    - BGP determines the routing of information



Delay Attack

- Can delay delivery of block up to 20 minutes
- Victim unaware of most recently mined block and its corresponding transactions
  - Double Spending attack
  - Wasteful mining



Questions?