CS578 – Blockchain Technology: A Software Engineering Perspective

Mid-Semester Assignment

Name: M. Maheeth Reddy	**Roll No**: 1801CS31	**Date**: 24-Feb-2022

---

1. What is formal method? Identity two issues in blockchain where formal methods play crucial roles (illustrate clearly with suitable example in each of these two cases)

Ans 1:

Edsger W. Dijkstra once said, "Program testing can be used to show the presence of bugs but never to show their absence!". This is where Formal Methods come into the picture.

Formal methods are mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems. They exploit the power of mathematical notation and mathematical proofs.

In other terms, they use math to analyze a program for error in logic and specify the errors. In the case of blockchain technology, formal verification methods could assure developers that smart contracts will work as intended, eliminating some of the bugs and financial losses that come as a result of coding errors.

Issues in Blockchain where formal methods are crucial:

Attack on The Decentralized Autonomous Organization:

The infamous DAO Attack happened on the Ethereum Blockchain on 17th June 2016. Around 3.6 million Ether had been stolen by the attackers (equivalent to $50 million at the time) by exploiting the re-entrancy problem in Smart Contracts.

When the DAO had sufficient crowdfunding, community members were allowed to withdraw a certain amount of ether from its contract via its Ethereum address. According to the ether transfer protocol, the ether would be sent to the address of the receiver before the balance was updated in the contract's internal state.

So, the attackers created a new contract and invoked the transfer function from the DAO contract. Since, the DAO's internal state was not being updated, different invocations of the function allowed the attackers to withdraw more ether than they were eligible for.

This is called a re-entrancy attack. We can use formal methods to prevent this by making sure that the internal state of the contract (which in case of DAO is the contract balance), is modified before any recursive calls take place. Interaction with other contracts should be the very last step. An additional measure to prevent the attack is to reduce the amount of gas passed to the transfer function.

Parity Wallet Hack on Ethereum:

In 2017, a vulnerability was found in the smart contract of the Parity Multi-sig Wallet, that allowed an attacker to steal over 150,000 ETH (approximately 30M USD).

Multi-sig wallets are like regular wallets in that they are also smart contracts, but require multiple approvals to withdraw any amount out of the wallet. So, one could deploy a multi-sig wallet contract with 4 people and set a rule that you need at least 3 of the 4 to approve any withdrawals from the wallet.

The attacker sent the following two transactions to the affected Multi-sig contracts:

The First one: To obtain exclusive ownership of the Multi-sig Wallet
The Second one: To move all of its funds.

The first transaction was performed by calling a constructor function called initWallet in the wallet contract. This function initializes the owner of the wallet contract.

The wallet contract also had a function that forwarded all unmatched function calls to delegatecall, a library function. When that function was called, the initWallet became publicly callable, despite being a constructor. So, the attacker exploited this and changed the ownership of the contract to himself.

So, to draw funds from the contract the attacker just needed to call a transfer function to get hold of all the funds in the contract. This execution was automatically authorized since the attacker was then the only owner of the multi-sig.

By using formal methods, we can test the smart contract on all possible inputs and scenarios in lesser time. So, further time could also be spent on checking the behavior of the contract when it is being called from other contracts. This could have revealed that the constructor initWallet had no checks to stop an attacker from calling it after the contract was initialized.

Conclusion: It is very much necessary that all kinds of bugs must be ironed out before the smart contracts are deployed because the most innocent-looking bug can have disastrous consequences.

As the application of formal methods to improve smart contract security is on the rise, developers can use automated theorem provers like Isabelle, Coq, Metamath that can check or even partially construct a formal proof.

2. Explain how the quorum size of (2f + 1) guarantees safety and liveness in PBFT consensus algorithm, where f denotes the number of Byzantine nodes.

Ans 2:

I will explain that a quorum size of (2f+1) guarantees safety and liveness in the PBFT consensus algorithm by proving the statement mathematically.

Given: The number of Byzantine nodes in a network is f.

To prove: In PBFT consensus, quorum size of (2f+1) guarantees the safety and liveness of the network.

Proof:

It is given that there are f Byzantine nodes in the network.

Quorum is defined as the minimum number of nodes needed for the network to run properly and make valid decisions. It only consists of honest nodes.

Let us assume there are totally N nodes in the given network and Q is the Quorum size.

Now, we need to find out the value of Q required to guarantee Liveness and Safety in the given network.

To maintain Liveness in the network, we need to avoid the network from stalling. So, there must be at least one non-faulty node. Therefore,

$$Q \leq N - f \quad \text{------- (1)}$$

In order to avoid the network from splitting into multiple decisions, the majority should be present. In a non-Byzantine attack scenario, the Quorum size should be greater than half of the total number of nodes. So,

$$2Q - N > 0$$

But in Byzantine failure, we need to consider the faulty nodes in the voting process as well because they can also participate in the voting process. So, the inequality changes to:

$$2Q - N > f \quad \text{---------- (2)}$$

If we combine inequalities (1) and (2), we get:

$$N+f < 2Q <= 2(N-f) \quad \text{-------- (3)}$$

Now consider, the highlighted portion of inequality (3). We get,

$$N+f < 2(N-f)$$

$$=> N+f < 2N - 2f$$

$$=> 2f+f < 2N - N$$

$$=> N > 3f$$

So, the minimum possible value of N is $N_{min} = 3f+1$.

Since, we have already shown $2Q > N+f$ in inequality (3), 2Q will be greater than $(N_{min}+f)$ also. Therefore,

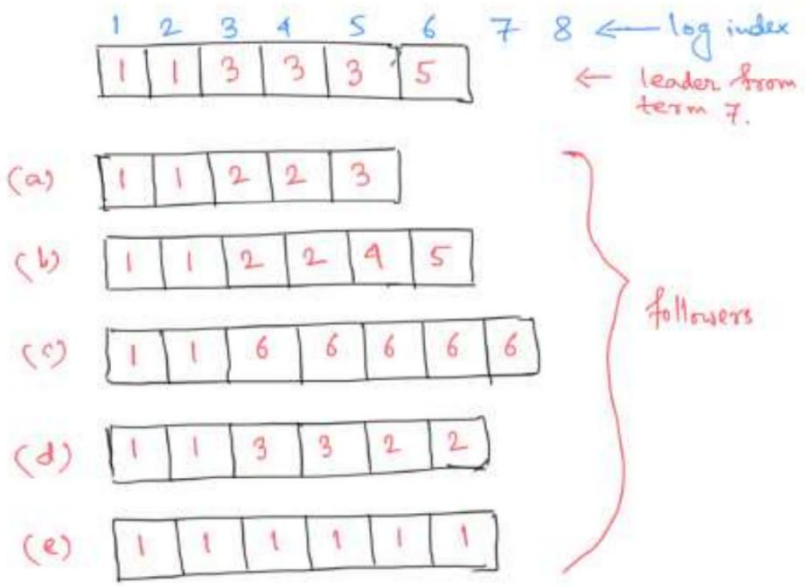
$$2Q > 3f+1 + f$$

$$=> Q > 2f + ½$$

So, minimum value of Q is $Q_{min} = 2f + 1$

Hence, we have proved that the quorum size of (2f + 1) guarantees safety and liveness in PBFT consensus algorithm, where f denotes the number of Byzantine nodes.

3. Consider a distributed system consisting of six nodes, which is using Raft consensus algorithm. The log status (below only log index and leader-terms are mentioned for simplicity) of the nodes are shown below. Provide your justification whether this log configuration for EACH of the followers is valid or not (justification is mandatory).



Ans 3:

Follower (a)

The log configuration is **INVALID**

Reason: The leader and follower (a) have the same term (term 3) at index 5. One of the Log Matching Properties of Raft Consensus is, "if two entries in different logs have the same index and term, then the logs are identical in all preceding entries".

But the preceding entries of follower (a) and leader are different:

For follower (a), the preceding entries are 1,1,2,2
For the leader, the preceding entries are 1,1,3,3

This contradicts Raft's property. So, the log configuration of follower (a) is invalid.

Follower (b)

The log configuration is **INVALID**

Reason: The leader and follower (b) have the same term (term 5) at index 6. One of the Log Matching Properties of Raft Consensus is, "if two entries in different logs have the same index and term, then the logs are identical in all preceding entries".

But the preceding entries of follower (b) and leader are different:

For follower (b), the preceding entries are 1,1,2,2,4
For the leader, the preceding entries are 1,1,3,3,3

This contradicts Raft's property. So, the log configuration of follower (b) is invalid.

Follower (c)

The log configuration is **INVALID**

Reason: Follower (c)'s log entry shows term 6 at index 3. Meanwhile, the other nodes have either term 1 or 2 or 3.

If we follow the Restrictions on Leader Election in the Raft consensus algorithm, the last term in the server is larger than the last term in the candidate.

So, follower (c) is forbidden from the leader election. It will not receive or cast any votes due to which it will not get leadership in term 6 as depicted. So, the log configuration of follower (c) is invalid.

Follower (d)

The log configuration is **INVALID**

Reason: In the Raft consensus, the term numbers will only increase monotonically. Term 2 will start only after term 1 has ended, term 3 starts only after term 2 has ended, and so on.

Observe indexes 4 and 5 in the log configuration of follower (d). We can notice that term 2 has started after term 3. This is contrary to Raft's property. Hence the log configuration of follower (d) is invalid.

Follower (e)

The log configuration is **VALID**

Reason: We can observe that follower (e)'s log configuration has extraneous entries. And extraneous entries are completely valid. Extraneous entries are extra and invalid entries in the logs of a follower. It is possible that the first leader may have been cut off from the network multiple times, due to which the entries are as shown. Or, there might have been multiple crashes or partitioning of nodes during previous terms. Now, the new leader must make follower (e)'s logs consistent with its own entries.

----------x----------x----------