Security Analysis on dBFT protocol of NEO

Qin Wang[1], Jiangshan Yu**[2], Zhiniang Peng[3], Van Cuong Bui[1], Shiping Chen[4], Yong Ding[5], and Yang Xiang[1]

[1] *Swinburne University of Technology*, Melbourne, Australia
{qinwang,vancuongbui,yxiang}@swin.edu.au
[2] *Monash University*, Melbourne, Australia jiangshan.yu@monash.edu
[3] *Qihoo 360 Core Security*, Beijing, China pengzhiniang@360.cn
[4] *Csiro, Data61*, Sydney, Australia Shiping.Chen@data61.csiro.au
[5] *Cyberspace Security Research Center*, Shenzhen, China stone_dingy@guet.edu.cn

Abstract. NEO is ranked as one of the top blockchains by market capitalization. We provide a security analysis on its backbone consensus protocol, called delegated Byzantine Fault Tolerance (dBFT). The dBFT protocol has been employed by NEO and other blockchains like ONT. dBFT claims to guarantee safety when no more than $f = \lfloor \frac{n}{3} \rfloor$ nodes are Byzantine, where n is the total number of consensus participants. However, we identify attacks to break the safety with no more than f Byzantine nodes. This paper provides the following contributions. First, we evaluate NEO's source code and present the procedures of dBFT. Then, we present two attacks to break the safety of dBFT protocol with no more than f nodes. Therefore, the system cannot guarantee the claimed safety. We also provide recommendations on how to fix the system against the identified attacks.

Keywords: Blockchain · NEO · dBFT · Safety.

1 Introduction

NEO has been one of the top-ranked blockchain platforms by its market capitalization. Rebranding from the Antshares in June 2017, NEO becomes the earliest and the longest-running public chain in China. From about 0.1 USD at the beginning of 2017, NEO reached a value of 160 USD at the end of 2017. At the time of writing, it's market capitalization is about 0.67 billion USD[6]. The thousandfold return on the investment placed NEO in the ranks of top blockchains within China and abroad. NEO has successfully established a matured ecosystem with decentralized applications (DApps), including games, lotteries, wallets, and exchanges. Furthermore, NEO has developed a complete architecture covering the consensus mechanism and components including NeoX, NeoFS, NeoQS [29] [20]. As the core protocol, dBFT was later adopted by the Ontology platform as one of the pluggable consensus mechanisms [23].

** Corresponding author.

[6] https://coinmarketcap.com/currencies/neo/. Data fetched on 21st Sept. 2019.

Consensus protocols make distributed participants collectively reach an agreement, which enables the immutability and prevents the forks within blockchain systems. Byzantine fault tolerance (BFT) consensus and its variants (together denoted as BFT-style consensus) tolerate a certain number of Byzantine participants who can misbehave. BFT-style mechanisms are permissioned, and provide a deterministic consensus guarantee [25] [26]. Various projects [18] employ the BFT-style consensus for their special needs. In particular, Practical Byzantine Fault Tolerance (PBFT) [9] is used as the foundation of many variants, as it enables the system to efficiently (with polynomial complexity) tolerate participants with arbitrary faults. For example, a variant of PBFT has been implemented for Hyperledger Fabric v0.5 and v0.6 [3] and Hyperledger Sawtooth v1.0 [4] [7].

dBFT is also a variant of PBFT, with the modifications on network model (from Client/Server to P2P), rule of permission (from fixed to dynamic) and procedure of commit (from 3-phase to 2-phase). dBFT focuses on the performance and scalability, however, the security has not been seriously analyzed. A comprehensive security analysis is absent from the official documents, including its whitepaper [20], documentation [29], and GitHub documents [19]. In fact, after examining the source code, we find that the implemented protocol is slightly different from what has been presented in the whitepaper. For example, in the official presentation of the protocol, not all messages transferred are signed, while in the actual implementation they are all signed and should provide a better security guarantee.

To evaluate the security of NEO, we first analyze the source code and provide a formal and accurate presentation of dBFT with the security goals. Then, we proposed two attacks against dBFT. Both attacks are on the safety of dBFT, making conflict decisions possible. This violates the agreement property where all honest replicas should agree on the same decision. Both identified attacks need to require a view change to happen. The first attack assumes a malicious primary to trigger the view change and the second attack requires a timeout (when the network asynchrony makes a quorum unavailable) to trigger the view change. Both attacks only require no more than $f = \lfloor \frac{n}{3} \rfloor$ malicious replica, where n is the total number of consensus participants and f is the number of Byzantine nodes that the system is supposed to tolerate. We also provide recommendations on fixing the identified vulnerabilities. Our contributions are summarized as follows:

- We provide the first clear presentation of the widely adopted dBFT consensus mechanism, based on its source code [5]
  `git commit 5df6c2f05220e57f4e3180dd23e58bb2f675457d`.
- We identify two attacks on dBFT. Both attacks are feasible with no more than $\lfloor \frac{n}{3} \rfloor$ nodes, where the first attack requires the primary to be Byzantine, and the second attack requires a timeout of the current view.
- We provide recommendations to fix the identified problems.

The rest of our paper is structured as follows: Section 2 provides an overview of PBFT. Section 3 defines the network assumption and the security properties, and Section 4 provides the detailed dBFT protocol, with a comparison with

PBFT protocol. Our identified attacks are presented in section 5. We provide the recommended fix in section 6, and the related work in section 7. Finally, Section 8 concludes the paper.

Communication with NEO. We have fully disclosed our results, including both identified vulnerabilities and the recommended fixes, to the NEO team. They acknowledge that our attacks are valid on their system, and have applied the fixes [1] [2].

2 Overview of PBFT

Practical Byzantine Fault Tolerance (PBFT), proposed by Castrol [9], is the most prevailing BFT consensus mechanism employed by current permissoned blockchain systems. It enables a system to efficiently (with polynomial complexity) tolerate $f = \lfloor \frac{n}{3} \rfloor$ malicious nodes out of the total n nodes. PBFT is designed in the partially synchronized network model, and proceeds in rounds denoted as *view*. There are three entities contained in PBFT: *Client*, *Primary* and *Replica* and three phases involved in the protocol: *Pre-Prepare*, *Prepare* and *Commit*. We follow the descriptions of [9] and [22], and the communication pattern of PBFT protocol is shown in Fig 1.

In the *Pre-prepare* phase, upon receiving a `REQUEST` message from a client, the primary node creates and broadcasts the `PRE-PREPARE` message to all the replicas. In the *Prepare* phase, each replica checks the validation of the received `PRE-PREPARE` message. If the message is valid, the replica creates and sends a `PREPARE` message to all nodes. In the *Commit* phase, upon receiving validated `PREPARE` messages from a quorum (i.e., $2f + 1$ replicas), this node creates and broadcasts a `COMMIT` message to all nodes. The last step is to reply to the client about the result. If a node receives a quorum `COMMIT` messages from $2f + 1$ different nodes, then it executes the client request, creates and sends the reply to the client. A client accepts the reply if it receives a reply from at least $f + 1$ nodes. The *Pre-prepare* phase is a one-to-all communication, while the *Prepare* and *Commit* phases are all-to-all communications.

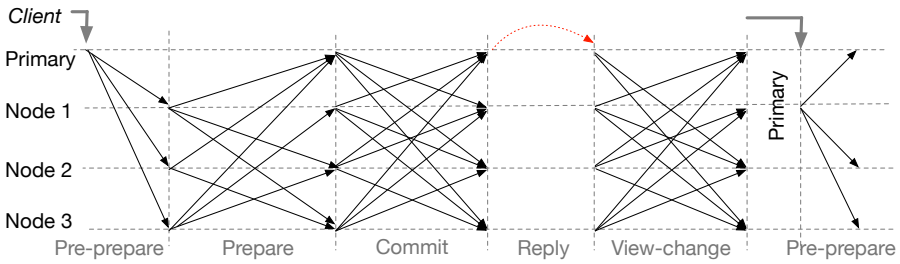


Fig. 1. PBFT Protocol

The primary is changed through a *View-change* protocol, if only if the primary is faulty or if network asynchrony breaks the availability of a quorum. In this case, the current round (view) is terminated and nodes initiate a view-change to update the primary. *View-change* makes a new primary node select from other nodes, and requires it to propose and send `NEW-VIEW` message containing the changed request under the same sequence number. After that, it enters the new view and continues the protocol.

3 Security Property

Safety and liveness are the main properties of a BFT protocol. The safety property requires that a "bad" event in the system will never happen, and the liveness property states that a "good" event will eventually happen. For example, PBFT guarantees safety when no more than $f = \lfloor \frac{n}{3} \rfloor$ are malicious, where n is the total number of nodes running PBFT. PBFT guarantees liveness when no more than f nods are malicious and the network is partial synchrony.

Network Assumption. Similar to PBFT, dBFT assumes a partially synchronous network [11], where a message sent from an honest node will eventually arrive within a fixed time-bound, but the bounded is unknown.

Security and liveness. While safety guarantees that the system behaves like a centralized implementation to maintain a total order sequence of decisions, liveness guarantees that clients eventually receive replies to their requests [9]. As a variant of PBFT, dBFT aims at providing the same guarantee under the same assumption – the safety is guaranteed when no more than f nodes are malicious, and the liveness is guaranteed with an additional assumption of a partially synchronous network.

4 dBFT Protocol

This section presents how dBFT works and its comparison with PBFT. Our presentation is based on the NEO official source code [19] and its technical reports [29] [20]. We summarize the detailed procedures and provide the call function workflow in the Appendix. Note that, to make it easier to understand, we adapt the terms used in PBFT to present dBFT.

4.1 Overview of dBFT

Entities in dBFT. dBFT has three types of nodes, called "speaker", "delegates" and "common nodes", and these types of nodes can be considered as the *Primary*, *Replica* and *Client* in the PBFT protocol, respectively. In dBFT, the primary node is randomly selected from the replicas to generate and send messages (proposals/blocks). The replicas are required to vote for the received messages

and maintain the globally ordered sequence of decisions (ledgers/blocks). They are selected from clients according to their reputation as defined by NEO. The client helps to disseminate messages through the underlying peer-to-peer network. They provide various end-user services including payment, exchange, and smart contracts.

State Transition in dBFT. There are three phases in the dBFT protocol, namely "Prepare", "Response" and "Publish". The former two phases serve for the consensus decision where the "publish" is used to broadcast the replies to a request. In particular, the "Prepare" and "Response" phases are similar to the "Pre-prepare" and "Prepare" phases in PBFT, respectively. The "Publish" is similar to the "Reply" step of the PBFT. For simplicity, we will use the terms defined in the PBFT to present the dBFT protocol, as they have been well accepted for decades.

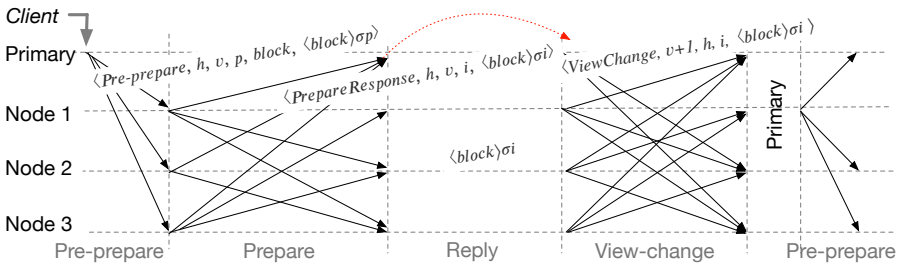


Fig. 2. dBFT Protocol

As shown in Fig 2, upon receiving the requests from a client, the primary starts the *Pre-prepare* phase by sending the `PRE-PREPARE` message to all replicas. Each replica verifies the validity of the received message. If valid, then it broadcasts a `PREPARE` message as its response. If a node receives `PREPARE` messages from a quorum ($2f+1$ nodes), then it executes the request and broadcasts its reply as its final decision. If the primary fails, dBFT runs its *View-change* protocol to reset the parameters and rotate the primary node.

4.2 Detailed Procedures of dBFT

Each execution of the dBFT protocol is initiated by its committee selection algorithm and leader election algorithm to form a consensus group and to select a primary from the group. When a primary and a consensus group is defined, the actual consensus execution protocol contains two main phases, namely *Pre-prepare* and *Prepare*. It also contains a *View-change* protocol when the primary is faulty or when the network asynchrony breaks the availability of a quorum.

Let h be the current block height (i.e., the length of the blockchain). Each replica is labeled by an index number i where $i \in [0, n-1]$ and n is the size of

the consensus group. At the beginning of each round, the primary p is selected from the consensus group following the rules of $p = (h - v) \mod n$. To reach an agreement on a block proposed by the primary node, each replica collects $2f$ signatures on the proposed block from other replicas, where $f = \lfloor \frac{n-1}{3} \rfloor$ is the assumed maximum number of Byzantine nodes. Once the agreement is reached, a new round of consensus begins, and the view is reset to $v = 0$. The block signed by replica i is defined as $block_{\sigma_i}$. Here we give the detailed procedures of each step, and the corresponding call function chart of each step can be found in the Appendix.

- *Committee selection:* The replicas (i.e., consensus committee members) are selected from the clients by the NEO foundation according to their reputation. Therefore, we omit the exact process here and put our focus only on the consensus algorithm.
- *Leader election:* The primary is determined by $(h - v) \mod n$, based on the current block height h, current view v and the size n of the consensus group. The leader rotates in the committee due to increased h.
- *Pre-prepare:* The primary creates a block containing valid transactions collected from the network, and sends a signed pre-prepare message $< \texttt{PRE-PREPARE}, h, v, p, block, < block >_{\sigma_p} >$ to all replicas.
- *Prepare:* After receiving the pre-prepare message, replica i checks the correctness of the message, including the validity of signatures, the correctness of h, v, p, block and the contained transactions. If the received proposal is valid, then it broadcasts a signed prepare message $< \texttt{PREPARE}, h, v, p, i, block, < block >_{\sigma_i} >$ to all replicas.
- *Reply:* After collecting signed and validated `PREPARE` messages from a quorum, the replica i is convinced that consensus is reached, and executes the request and broadcasts its reply $< \texttt{REPLY}, h, v, m, i, < block >_{\sigma_i} >$.
- *View-change:* When detecting a faulty primary or when a quorum is not available, the replica i sends a `VIEWCHANGE` message $< \texttt{VIEWCHANGE}, h, v + 1, p, i, block, < block >_{\sigma_i} >$ to other nodes. *View-change* is triggered when valid messages are received from a quorum.

4.3 Comparison with PBFT

dBFT is a variant of PBFT protocol with several modifications, as follows. In terms of protocol phases, dBFT removes several sub-protocols of PBFT. In particular, it removes the core *Commit* phase from the PBFT, and also removes the auxiliary protocols including *GarbegeCollection* and *Checkpoint*. In terms of the communication model, dBFT employs a peer-to-peer network topology to disseminate messages, rather than the previous client-server communication model. In terms of the message authentication, dBFT uses digital signatures to authenticate messages rather than using MAC as in PBFT. In terms of consensus committee, there are several changes. First, dBFT does not have a fixed consensus group as in PBFT. Rather, it implements a mechanism to enable dynamic joining/leaving of nodes to offer flexibility. Second, for leader election,

dBFT enforces the change of primary for each round of consensus. In particular, at the beginning of each consensus round, the new primary p is determined by $p = (h - v) \mod n$. So, whenever a new block is accepted in the blockchain, the primary will be changed.

5 Identified Attacks

This section presents two identified attacks on the safety of dBFT. Both attacks need to enforce a view change. The first attack requires a malicious primary to trigger the view change and the second attack requires a timeout (when the network asynchrony makes a quorum unavailable) to trigger the view change. Both attacks only require no more than f malicious replica, which is the case the dBFT is supposed to tolerate. We make use of a simple scenario with four nodes to demonstrate our attacks. Let $n = 4$, so $f = 1$. Let A_i be the identity of the i-th replica, where $i \in [0, n-1]$.

5.1 Attack Case 1

Let A_0 be the Byzantine node, and it is selected as primary. The detailed attack process is shown as follows.

- *step 1:* The Byzantine primary A_0 creates two blocks, *block1* and *block2*, such that they contain conflict transactions for e.g. spending a coin multiple times. A_0 then sends `<Pre-prepare>` on *block1* to A_1 and A_2, and sends `<Pre-prepare>` on *block2* to A_3.
- *step 2:* As both blocks are valid, A_1 and A_2 will create and broadcast a `<Prepare>` message on the *block1*, and A_3 will broadcast a `<Prepare>` message on *block2*.
- *step 3:* Since no replica receives enough valid `<Prepare>` message $(2f + 1)$ from a quorum, the current round will timeout, and it triggers the view change protocol.
- *step 4:* Run view change protocol honestly. Since in the previous view ($v = 0$), $(h - 0) \mod 4 = 1$, so in this view $v = 1$, A_3 will be elected as the primary, i.e. $(h - 1) \mod 4 = 3$.
- *step 5:* Run the consensus on *block2* with $v = 1$. When a decision is reached, A_0 can create a conflict decision by releasing $2f + 1 = 3$ valid `<Prepare>` messages on *block1* of view $v = 0$. This breaks the consensus safety.

5.2 Attack Case 2

Attack case 2 considers the scenario where the Byzantine replica is not primary for the current view, and it relies on the view change triggered by network asynchrony. (Note that unlike liveness, the safety should hold under network asynchrony.)

- *step 1:* Select the leader according to $p = (h - v) \mod n$.
- *step 2:* The honest leader sends a valid proposal `<Pre-prepare>` on *block1*.
- *step 3:* the Byzantine replica performs the following strategy. If it receives $2f + 1 = 3$ signed `<Prepare>` messages from others, it runs the protocol honestly. If it only receives two signed messages, then it does not react. This can happen due to network asynchrony. In the second case, there is a possibility that replicas timeout the current view, and request a new view.
- *step 4:* If a view change is triggered, then the Byzantine replica runs it honestly.
- *step 5:* If the Byzantine replica is selected as primary, then it proposes a valid proposal `<Pre-prepare>` on *block2*, which contains transactions conflict with the ones contained in *block1*.
- *step 6:* all nodes run the consensus protocol, and reach a decision on *block2* with the current view number. When the decision is reached, the Byzantine replica releases the two signed `<Prepare>` messages on *block1* collected in the previous view, together with its signed `<Prepare>` message also on *block1*. This creates a conflict decision and breaks the consensus safety.

6 Recommended Fix

As shown in the previous section, the safety of dBFT cannot be guaranteed even when no more than f replicas are malicious, as conflict agreements can be reached. Our identified attacks are in fact not new. It is known that it is possible to have a secure two-phase protocol for crash fault tolerance (CFT) protocols, but a two-phase PBFT is vulnerable against Byzantine replicas. Thus, the *Commit* phase becomes necessary [13], [9]. The fix then becomes straight forward – the *Commit* phase is necessary to guarantee the safety, and dBFT needs to add this phase back to make the protocol secure against the two identified attacks.

The *Commit* phase plays a role to check if at least $2f + 1$ replicas have responded to the request. If a node has collected $2f + 1$ signed responses in the *Prepare* phase, then it commits the block by signing it together with state information, and sends it to all replicas. If at least $2f + 1$ valid commits messages are collected, then the replica updates the local state of the blockchain by including the block into it, and broadcasts the result to the network. As this is a standard construction in the classic BFT protocol, and is proved to be secure [21], we omit the formal proofs in this paper.

7 Related Work

The consensus problem can be traced back in early 1975, when the Two Generals Problem with its insolubility proof was proposed [6]. The problem was formally proved to be unsolvable, providing a base of realistic expectations for any distributed consistency protocols. The *FLP* impossibility result [13] placed an upper bound on what it is possible to achieve with distributed processes in an asynchronous environment. The *CAP* [14] theorem states that distributed

systems cannot satisfy all three conditions, namely consistency, availability, and partition tolerance. BFT protocols can tolerate at most $f \leq \lfloor \frac{3}{n} \rfloor$ Byzantine nodes, unless a trusted component is used [24].

Bitcoin. Bitcoin [17] is a cryptocurrency introduced in 2008. It aims at tolerating $< 50\%$ malicious power in the system. Unlike traditional consensus protocols, it does not require a pre-fixed consensus group. Instead, it allows any node to join and leave the system. It makes use of a public ledger (a.k.a. a blockchain) to record all transactions in the system. The public ledger is a chain of blocks, where each block contains a sequence of transactions that have not been recorded in previous blocks. Everyone can read the ledger from the Bitcoin network, and can write on it by finding a block such that the hash value of the block is small enough. The process of finding a valid block is called "*proof of work*". This concept defeats Sybil attacks, where an attacker can create many fake nodes at a low cost. Different participants may create conflicting blocks. To provide consensus on the conflicting blocks, participants only accept the longest chain. However, this way of agreeing about blocks only provides a probabilistic guarantee, as it is possible for malicious participants to work on a short chain to race with a longer one, until the shorter one beats the longer chain. This leads to attacks such as double spending attacks [30] and selfish mining attacks [12]. In addition, the block size is currently limited to 1 MB. This limits its transaction throughput to 7 transactions per second, whereas other existing payment systems handle way more. For example, Visa confirms a transaction within seconds, and processes 2k TPS on average, with a maximum rate of 56k TPS. For more detail, we refer readers to a detailed comparison [25] between Bitcoin and BFT protocols.

Adapting BFT protocols in blockchain. Classic BFT protocols provide a better throughout and security guarantee. PBFT [9] proposes the first practical Byzantine fault-tolerant algorithm with acceptable performance. Zyzzyva [16] is a speculation-based BFT protocol that reduces cryptographic over-heads and increases peak throughput for demanding workloads compared to traditional state machine replication. However, an attack [15] on the safety of Zyzzyva has been identified. MinBFT/MinZyzzyva [24] proposes to use a trusted component to improve the performance and security of PBFT and Zyzzyva.

However, these systems cannot be adapted directly in the blockchain, as they require a pre-fixed consensus group. Many systems (e.g. [22, 27, 28]) have been proposed to adapt BFT protocols to address the shortcomings of Bitcoin blockchain. PeerCensus [10] was the first blockchain to propose using proof-of-work for selecting consensus committees, and use a BFT-style protocol for reaching consensus. dBFT takes a different approach, where the consensus committee is defined by NEO based on the social reputation of nodes. We refer readers to existing comprehensive surveys [8,18,26] on the membership selection algorithms, blockchain consensus, and identified attacks.

8 Conclusion

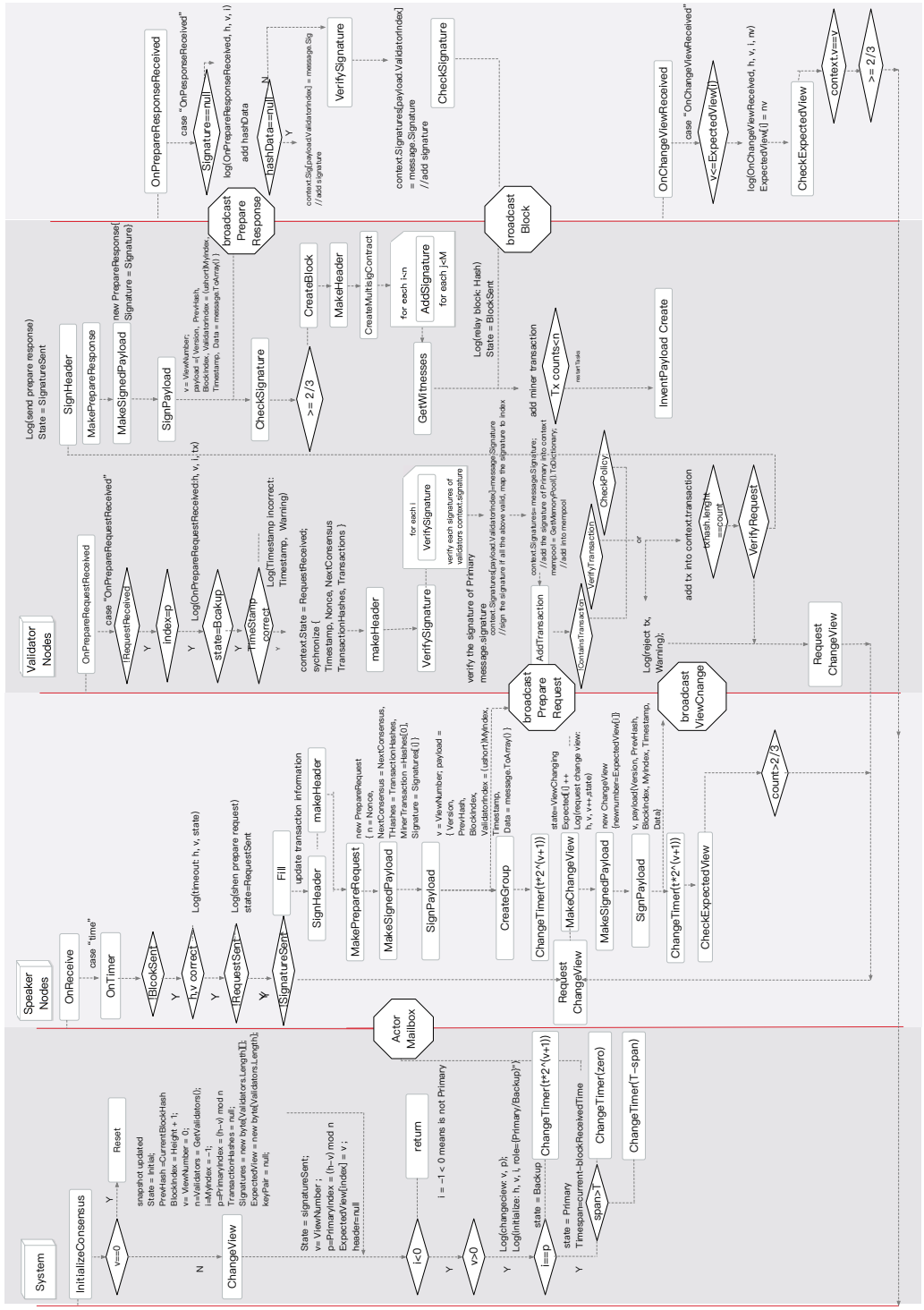
NEO, as the pioneer of public blockchain projects around the world, confronts severe security threats. Our security analysis is focusing on the core component of NEO, i.e., its dBFT consensus. As a variant derivative of PBFT, the dBFT consensus removes the important *Commit* processes compared to the original ones, resulting in deterministic forks under the specific conditions. In fact, it is known that removing the commit phase would lead to insecurity. This paper provides a study to revisit this issue, as a lesson learned from the already deployed and widely adapted consensus algorithm.

References

1. Discussion and improvement on dbft (2019), `https://github.com/neo-project/neo/pull/320`
2. Discussion and improvement on dbft (2019), `https://github.com/neo-project/neo/pull/547`
3. Hyperledger fabric (2019), `https://cn.hyperledger.org/projects/fabric`
4. Hyperledger sawtooth (2019), `https://cn.hyperledger.org/projects/sawtooth`
5. Neo source code on github (2019), `https://github.com/neo-project/neo/tree/master/neo`
6. Akkoyunlu, E.A., Ekanadham, K., Huber, R.V.: Some constraints and tradeoffs in the design of network communications. SIGOPS Oper. Syst. Rev. **9**(5), 67–74 (Nov 1975). https://doi.org/10.1145/1067629.806523
7. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference. p. 30. ACM (2018)
8. Cachin, C., Vukolić, M.: Blockchain consensus protocols in the wild. arXiv preprint arXiv:1707.01873 (2017)
9. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999. pp. 173–186 (1999). https://doi.org/10.1145/296806.296824, `http://doi.acm.org/10.1145/296806.296824`
10. Decker, C., Seidel, J., Wattenhofer, R.: Bitcoin meets strong consistency. In: Proceedings of the 17th International Conference on Distributed Computing and Networking. p. 13. ACM (2016)
11. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) **35**(2), 288–323 (1988)
12. Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. Communications of the ACM **61**(7), 95–102 (2018)
13. Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985). https://doi.org/10.1145/3149.214121, `https://doi.org/10.1145/3149.214121`
14. Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002). https://doi.org/10.1145/564585.564601, `https://doi.org/10.1145/564585.564601`

15. Ittai Abraham, Guy Gueta, D.M.J.P.M.: Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma (2018), https://arxiv.org/abs/1801.10022
16. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative byzantine fault tolerance. In: ACM SIGOPS Operating Systems Review. vol. 41, pp. 45–58. ACM (2007)
17. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008), https://bitcoin.org/bitcoin
18. Natoli, C., Yu, J., Gramoli, V., Esteves-Verissimo, P.: Deconstructing blockchains: A comprehensive survey on consensus, membership and structure (2019)
19. NEO: Neo github (2018), https://github.com/neo-project
20. NEO: Neo whiteopaper (2018), http://docs.neo.org/zh-cn/whitepaper.html
21. Rahli, V., Vukotic, I., Völp, M., Veríssimo, P.J.E.: Velisarios: Byzantine fault-tolerant protocols powered by coq. In: ESOP. pp. 619–650 (2018)
22. Stathakopoulou, C., David, T., Vukolić, M.: Mir-bft: High-throughput bft for blockchains. arXiv preprint arXiv:1906.05552 (2019)
23. Team, O.: Ont consensus (2018), https://github.com/ontio/ontology/tree/master/consensus/dbft
24. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Verissimo, P.: Efficient byzantine fault-tolerance. IEEE Transactions on Computers **62**(1), 16–30 (2011)
25. Vukolić, M.: The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In: International workshop on open problems in network security. pp. 112–125. Springer (2015)
26. Vukolić, M.: Rethinking permissioned blockchains. In: Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts. pp. 3–7. ACM (2017)
27. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: Bft consensus with linearity and responsiveness. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 347–356. ACM (2019)
28. Yu, J., Kozhaya, D., Decouchant, J., Veríssimo, P.J.E.: Repucoin: Your reputation is your power. IEEE Trans. Computers **68**(8), 1225–1237 (2019)
29. Zhang, E.: Neo consensus (2018), http://docs.neo.org/en-us/basic/consensus/consensus.html
30. Zhang, R., Preneel, B.: Lay down the common metrics: Evaluating proof-of-work consensus protocols security. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE (2019)

A dBFT Flow Chart


NEO DBFT Consensust Flow Chart with call functuin
System
InitializeConsensus
v==0
Y
Reset
snapshot updated
State = Initial;
PrevHash =CurrentBlockHash
BlockIndex = Height + 1;
v= ViewNumber = 0;
n=Validators = GetValidators();
f=(Myindex = -1;
p=PrimaryIndex = (h-v) mod n
TransactionHashes = null;
Signatures = new byte[Validators.Length][];
ExpectedView = new byte[Validators.Length];
keyPair = null;
N
ChangeView
State = signatureSent;
v= ViewNumber ;
p=PrimaryIndex = (h−v) mod n
ExpectedView[index] = v ;
header=null
i<0
Y
return
i = –1 < 0 means is not Primary
v>0
Y
Log(changeview: v, p);
Log(Initialize: h, v, i, role=[Primary/Backup]");
i==p
Y
state = Backup
ChangeTimer(t*2^(v+1))
state = Primary
Timespan=current–blockReceivedTime
span>T
Y
ChangeTimer(zero)
ChangeTimer(T–span)
Actor Mailbox
Speaker Nodes
OnReceive
case "time"
OnTimer
IBlockSent
Y
h,v correct
Y
Log(timeout: h, v, state)
RequestSign
Y
Log(shen prepare request)
state=RequestSent
SignatureSent
Fill
update transaction information
SignHeader
makeHeader
MakePrepareRequest
new PrepareRequest
(v, NextNo,
NextConsensus = NextConsensus,
THashes = TransactionHashes,
MinerTransaction =Hashes[0],
Signature = Signatured[])
MakeSignedPayload
SignPayload
v = ViewNumber; payload =
{ Version,
PrevHash,
BlockIndex,
ValidatorIndex = (uint)MyIndex,
Timestamp,
Data = message.ToArray() }
CreateGroup
ChangeTimer(t*2^(v+1))
Request ChangeView
MakeChangeView
state=ViewChanging
Expected[i] ++
Log(request change view:
h, v, v++,state)
MakeSignedPayload
new ChangeView
(newnumber=ExpectedView[i])
SignPayload
v, payload)Version, PrevHash,
BlockIndex, MyIndex, Timestamp,
Data)
ChangeTimer(t*2^(v+1))
CheckExpectedView
count>2/3
broadcast ViewChnage
Log(reject tx;
Warning);
or
add tx into context.transaction
if(hash length
==count)
VerifyRequest
broadcast Prepare Request
AddTransaction
ContainsTransaction
VerifyTransaction
CheckPolicy
context.Signatures = message.Signature;
// add the signature of Primary into context
mempool = GetMemoryPool().ToDictionary;
//add into mempool
add miner transaction
Tx counts<n
InventPayload Create
broadcast Block
Log(relay block: Hash)
State = BlockSent
Validator Nodes
OnPrepareRequestReceived
case "OnPrepareRequestReceived"
RequestReceived
Y
index=p
Y
Log(OnPrepareRequestReceived(h, v, i, tx)
state=Backup
Y
PrimeStamp
correct
Y
Log(Timestamp incorrect:
Timestamp, Warning)
context.State = RequestReceived;
sychronize {
Timestamp, Nonce, NextConsensus
TransactionHashes, Transactions }
makeHeader
VerifySignature
for each i
VerifySignature
verify each signatures of
validators context.signature
verify the signature of Primary
message.signature
//sign the signature if all the above valid, map the signature to index
Log(send prepare response)
State = SignatureSent
SignHeader
MakePrepareResponse
MakeSignedPayload
new PrepareResponse(
Signature = Signature)
SignPayload
v = ViewNumber;
payload =(Version, PrevHash,
BlockIndex, ValidatorIndex, (uint)MyIndex,
Timestamp, Data = message.ToArray())
broadcast Prepare Response
CheckSignature
>= 2/3
CreateBlock
MakeHeader
CreateMultiSigContract
for each i<n
AddSignature
for each j<M
GetWitnesses
OnPrepareResponseReceived
case "OnPrepareResponseReceived"
Signature==null
Y
log(OnPrepareResponseReceived, h, v, i)
add hashData
hashData==null
Y
VerifySignature
CheckSignature
context.Signatures[payload.ValidatorIndex]
= message.Signature
//add signature
OnChangeViewReceived
case "OnChangeViewReceived"
v<=ExpectedView[i]
log(OnChangeViewReceived, h, v, i, nv)
ExpectedView[i] = nv
CheckExpectedView
context.t==v
Y
>= 2/3