BFT Protocol Forensics

CS598FTD Report Spring 2022

Milind Kumar V
mkv4@illinois.edu
University of Illinois
Champaign, IL

Chirag Shetty
cshetty2@illinois.edu
University of Illinois
Champaign, IL

1 INTRODUCTION

In this report we explore the idea of forensics for BFT protocols. BFT protocols are designed to solve consensus among n replicas as long as the number of Byzantine replicas or faults f is less than a threshold t. If there are more faults than t, either safety or liveness can be violated. Safety is violated when atleast two honest replicas commit two different values. Forensics is a "day after" analysis, which asks, can we identify the Byzantine nodes responsible for this violation? More importantly, the analysis should be able to provide irrefutable proof of culpability of those nodes. In a real scenario, where we do not know which nodes are honest, forensics makes sense only if the honest nodes can put together a proof that shows the perpetrators of the protocol violation. Naturally, the proof can not depend on logs of all replicas, since Byzantine nodes can forge it. The best forensics would require logs of as few nodes as possible to prove culpability of as many Byzantine nodes as possible.

We describe parameters that characterize a forensics protocol in Section 2. This is followed by a discussion of the problem setting and the particular algorithm under consideration in Section 3. Section 4 discusses safety violations, forensics algorithms and the intuition for associated proofs. In Section 5, we discuss some unexplored ideas such as the notion of "commit" and finality, liveness violations, trust nodes and point out some hidden costs not considered in [4].

2 FORENSICS: PARAMETERS & FEASIBILITY

For BFT protocols, forensics comes into play after a safety violation has occurred. We wish to be able to identify as many malicious nodes as possible with an irrefutable cryptographic proof that identifies some set of malicious nodes as culpable. Further, we also wish to minimize the communication required between nodes after a violation has occurred during forensics. We can formalize these notions using the tuple (m, k, d) which describes the forensic support available for an algorithm.

- m: This is the maximum number of Byzantine faults the forensics algorithm allows before it can no longer detect $d > 0$ malicious nodes. For algorithms requiring a quorum of size $2t + 1$, this can reach at most $2t$. If there were greater than $2t$ Byzantine nodes, they would be able to simulate the algorithm without the participation of any honest nodes and it would be impossible to perform forensics.
- k: This is the minimum number of transcripts the protocol requires from honest nodes to guarantee the detection of faulty nodes. The lowest value this can take is 1 as at least a single transcript is required to guarantee that forensics can be performed. Note that in certain cases, such as when two nodes output different verifiable values in the same view in PBFT, it might be possible to identify the culpable nodes with 0 transcripts obtained after the attack. However, $k = 1$ indicates that it might not always be possible to do so for other types of attacks.
- d: This is the number of Byzantine replicas the forensics algorithm is guaranteed to identify. The maximum value this can take is $t + 1$. While a forensics algorithm might identify more than $t + 1$ nodes for particular safety violations, it can not be guaranteed to identify more than $t + 1$ malicious nodes as even if there are more, $t + 1$ nodes can cause a violation while the rest behave like honest nodes.

From the discussion above, it can be concluded that $(2t, 1, t + 1)$ is the strongest support that can be offered for algorithms solving SMR. This is achieved by PBFT-PK which is described in Section 3 and the corresponding forensic algorithm is presented in Section 4.

3 PBFT - RECAP

3.1 Problem setting

We consider the problem of state machine replication- a setting in which the goal is to provide clients a service with the illusion of a single non-faulty server despite some of the servers- called replicas or nodes henceforth- being faulty [3]. Each client must receive the same totally ordered sequence of values. In this work, we will focus on outputting a single value rather than a sequence of values. Further, our problem has the following three parameters:

- n: number of replicas participating the consensus protocol
- t: maximum number of faults the algorithm is designed to tolerate before the guarantees of Section 3.1.3 are violated
- f: the actual number of faulty nodes in a given run of the algorithm

3.1.1 Timing model. For this problem setting, we will consider the model of Partial synchrony. Partial synchrony strikes a balance between the Synchronous model where all messages, once sent, arrive within a known, finite time bound Δ and the Asynchronous model where an adversary can delay a message by an arbitrary amount of time but must deliver all messages eventually. In the Partial Synchrony model [1], there exist a known, finite time bound Δ and a special event called the Global Stabilization Time (GST henceforth) such that

- The adversary must cause the GST to happen after some time. However, the time after which the adversary chooses to make this happen is unknown.
- Any message sent at a time x must arrive by $\Delta + max(x, \text{GST})$. This means that any message sent after the GST must arrive within Δ time of being sent.

3.1.2 Fault model. We will consider Byzantine faults. Here, an adversary's actions can be arbitrarily malicious.

3.1.3 Properties. We are required to guarantee the following properties as long as the number of faulty nodes in operation are less than the maximum number of faulty nodes the algorithm is willing to tolerate i.e $f < t$:

- **Safety:** No two honest replicas will output different values
- **Liveness:** A value sent by the client will eventually be output by honest replicas
- **Validity:** We consider external validity here, i.e replicas only output signed values sent by the client

3.2 PBFT

The full PBFT-PK algorithm is provided in Algorithm 1. The PK indicates that digital signatures are used for all messages. Signatures allow forwarding of messages i.e node A can verify that a message that node B claims was sent to it by node C was indeed sent to it by node C. The absence of this feature is what makes it impossible to provide forensic support for the PBFT variant using MACs instead of signatures. Here, all to leader to all voting is used where messages are routed through the leader. Further, note that for the guarantees of Section 3.1.3 to hold, we require $n > 3t + 1$. The algorithm has several phases.

VIEW CHANGE In this algorithm, the nodes go through a series of views with the variable e being used to denote a view. Each view has a unique leader. A view change occurs when enough nodes timeout upon not receiving some desired message within the required interval. This ensures liveness in the partial synchrony setting- while it is impossible to distinguish if the leader is faulty or is experiencing network delay, it is always possible to elect a new leader and start over. For $f \leq t$, we show that a leader will never fail to propose a value previously committed by some honest node. The view change occurs as follows

- A node that has timed out broadcasts a BLAME message. It then collects $t + 1$ BLAME messages and broadcasts them. Since we assume that there are at most t faulty nodes, this ensures that at least one honest node has broadcast a BLAME message and that the new leader is justified in stepping up. Collecting and broadcasting $t+1$ BLAME messages also allows an honest node to begin timeout from that point on as it knows that it has provided the new leader with sufficient votes to step up.
- The node then exits this view, increments its view number and sends a STATUS message indicating its current locked value.

PRE-PREPARE phase In this phase, the leader (who can step up based on the view number) sends a PROPOSE message. This contains a status certificate M which consists of $n - t = 2t + 1$ STATUS messages, each indicating the value and view number the sending node is locked on. The leader picks the value corresponding to the highest view number from M. It is important that the leader collect $2t + 1$ STATUS messages. This ensures safety across views as we will see in the Commit phase.

PREPARE phase Upon receiving a PROPOSE message for value v, every node verifies that the leader indeed has sent a valid message by verifying that the every locked value in every STATUS message in M is valid and that the leader indeed picked the highest lock among the ones available to it. Once the PROPOSE message is verified, every honest node sends a VOTE1 for v to the leader.

COMMIT phase The leader, upon receiving $n - t = 2t + 1$ VOTE1 messages for a value v, aggregates them into a signature and broadcasts a COMMIT message with the value, view number and signature. The nodes upon receiving a valid COMMIT message lock on the value v and the current view number, say e. Note that for $f \leq t$ Byzantine nodes, it is not possible for two honest nodes to lock on different values in this phase as this would imply that each value received $2t + 1$ VOTE1s. This would in turn imply that $(2t + 1) + (2t + 1) - (3t + 1) = t + 1$ nodes double voted which is impossible for $f \leq t$.

Once a node locks on a value, it sends VOTE2 for the value.

REPLY phase The leader, upon receiving $n - t = 2t + 1$ VOTE2 messages for a value v, aggregates them into a signature and broadcasts a REPLY message with the value, view number and signature. Upon receiving a reply message, the nodes validate it and if correct, output the value v. The client is also provided the signature sent to the nodes by the leader along with the value v in order allow it to validate the output. Note that in order for two honest nodes to output two different valid values in the same view, each must have received at least $2t + 1$ valid VOTE2s which is impossible from the quorum intersection argument for $f \leq t$. This ensures safety within a view. During the view change, the leader collects $2t + 1$ statuses and proposes a new value. Suppose a new leader steps up in view e and collects STATUS messages and that a value v' was committed in the previous view $e - 1$. Now, a commitment in view $e - 1$ implies that at least $2t+1$ values locked on v' in view $e-1$. For $f \leq t$, at least $t+1$ of these locks are by honest nodes. This ensures that the status of at least one honest node $((2t + 1) + (t + 1) - (3t + 1))$ is included in those collected by the leader. Since the commit was made in the immediately previous view, it is the highest one and will be selected by the leader for the PROPOSE message. This ensures safety across views.

4 FORENSICS FOR PBFT

Now we consider the case when $f > t$ and a safety violation has occurred. f needs to be only 1 more than t to cause such a violation, and that is the cases we will consider. First we look at the mechanism of causing such a violation and then we present the forensics algorithm following from it. The complete algorithm is presented as Algorithm 2 in the Appendix.

4.1 What happens when f is more than t?

PBFT uses two key features to ensure safety when $f < t$ - quorum intersection and signatures. Quorum intersection primarily serves two purposes - prevent leader equivocation (in VOTE1) and ensuring a value that is possibly committed by some node is re-proposed across views. Signatures on the other hand ensure that each messages is backed up with a proof that the sender is following the protocol. Eg: proving reception of enough BLAMES before stepping in as the leader. Safety violation results from Byzantine nodes' ability to cause incorrect quorums as we see next. Signatures allow the forensics, where in each node can be held accountable for the messages it sent to cause the violation.

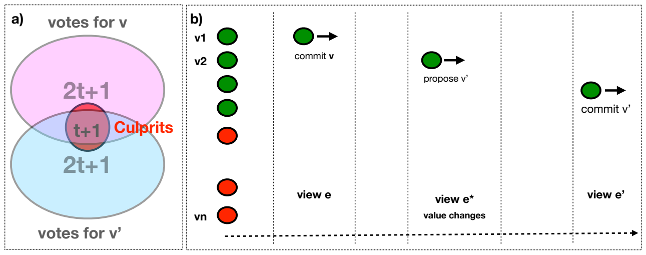

Figure 1: a) Violation within a view b) Timeline of violation across views

Safety violations can be of two kinds. Violation within the same view i.e two honest nodes commit two different values in the same view. Or violation can happen across views when an honest node commits a value and after one or many view changes another honest node commits a different value.

4.1.1 In a view. Refer to Fig. 1(a). Any node commits when it gets $2t+1$ VOTE2s. So if two honest nodes commit different values in the same view, they both got 2t+1 VOTE2s. This implies that atleast $t+1$ nodes vote for both the values, thus causing the violations. Since the messages are signed, the list of VOTE2s received by each of the commits will reveal the double-voters.
Algorithm: Consider lines 24 through 26 of Algorithm 2. A client receives two REPLY messages in the same view e with conflicting values v and v'. In this case, from the discussion above, at least $t+1$ culpable nodes can be found by finding the nodes who signed both the first and second REPLY messages. Note that no additional communication is needed here.

4.1.2 Across view. Analysing violation across views is slightly more involved. Refer to Fig. 1(b). A node may commit in a value v in view e and another node may commit value $v'(\neq v)$ in a future view e'. In PBFT, the view change protocol ensures safety by requiring the leader to collect $2t+1$ statues and choosing the latest locked value out of them. So a violation occurs if some nodes support commit of v in e by sending VOTE2 (and thus locking), but do not include that value in the status message they send to the new leader. They can do so by sending an old lock. The new leader may thus be unaware of the previously locked value and go on to propose one of the old locked values. However the key point is that this malicious behaviour may happen at some intermediate view e^* where the value change actually occurs. The forensics algorithm to identify Byzantine nodes when a safety violation occurs across views is as below

- Obtain the list of nodes that signed from the signature of the first REPLY message.
- Request a proof from the nodes participating in the consensus algorithm. Considering that two different values have been committed, it must be the case that a new value other than the original reply was proposed at some point in time. Every node looks through its transcript of PROPOSE messages to identify the view where that happened.
- Upon discovering such a view, the node provides the client with the PROPOSE message (lines 17 through 22 of Algorithm 2) where a new value was proposed. The intersection of the signers of the status messages of this proposed message and those from the first REPLY message yields at least $t+1$ malicious nodes.
- In case there are two locks that serve as the highest lock amidst the statuses of PROPOSE message identified by the node, $t+1$ malicious nodes can once again be detected from the intersection of the nodes that help form the two locks in view e''. This is shown in lines 18 and 19 of Algorithm 2.
- Note that obtaining one honest transcript is sufficient here.

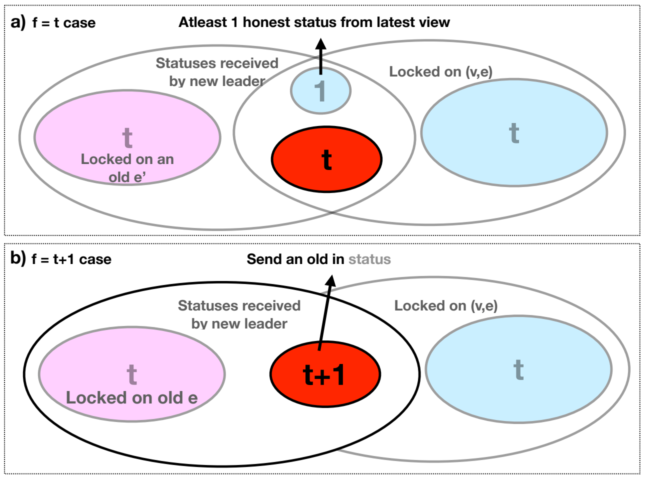

Figure 2: Mechanism of safety violation during view change

4.2 PBFT MAC: Impossibility Intuition

In PBFT MAC signing the messages is not required. Instead MACs ensure that the receiver of a message can be sure of who the sender is. If a violation occurs, even if the respective honest nodes can figure out the Byzantine nodes from its message logs, there is no way to create an irrefutable proof. For any proof that is generated, one could imagine a scenario where the honest node that created the proof itself is malicious and the accused are actually honest. The inability to 'forward' messages in MAC based scheme, unlike with signatures causes PBFT MAC to not have any forensic support.

5 COMMENTS

5.1 Attack difficulty & Continuous monitoring

Firstly, in BFT protocols we assume the strongest adversary, meaning the adversary can control the network and the order in which messages arrive. This allows us to perform a worst case analysis of the security of our algorithm. However, for academic interest, we can ask how easy the attack we discussed is in real life. And also consider what the best strategy is as an adversary (similar to the slightly modified private chain attack for Nakamoto consensus in Bitcoin).

In both cases 4.1.1 and 4.1.2, malicious nodes must get themselves into two quorums. To cause 4.1.1, all malicious nodes vote for every value they get without respecting the need to send only one VOTE1 or VOTE2 in a view. However for this to happen, two values must be present in the instantiation of the algorithm. This can happen when the protocol begins and the leader is malicious. Note that in PBFT, VOTE1 and VOTE2 need not be backed up with proofs. Thus malicious nodes are free to send votes for any value. However, to cause a violation, they must get a quorum. So they still need support of honest nodes (eg: if $f = t + 1$, a quorum still requires t honest nodes). But honest nodes will send out a vote only if the pre-requisites are met (eg: getting $2t + 1$ VOTE1 before sending VOTE2). Thus, the leader must be malicious and it must be the first view. Hence although 4.1.1 looks simpler, causing a violation within the same view requires a sophisticated attack.

In 4.1.2, the malicious nodes cause violation by omission. In order to cause the proposal of a new value in view e^* after the commit in e, they are required to collaborate and this incurs a communication cost. In order for an attack to succeed, at least $t + 1$ nodes have to 1) send VOTE2s to commit the value v in view e 2) cause a view change in e^* by manipulating network delay so that a lock with a view value $e'' \leq e$ is the highest lock a leader receives from the honest nodes 3) collude to send lower locks than e''. As is evident, this is quite a sophisticated attack.

Expectedly, chance of success increases as number of malicious nodes increases. Nevertheless, it is hard to pull off an attack and it requires continuous attempts. Meanwhile the Byzantine nodes leave a trail of malicious behaviour. A constantly monitoring forensics system (as was suggested in a question during our presentation) rather than a 'day-after' one discussed in the paper, may succeed in identifying the Byzantine nodes even before a violation happens. Forensics can also be built into the nodes, where in the honest nodes can flag Byzantine nodes. For instance, this can happen when a honest node sees two VOTE1 or VOTE2 messages from the same node for two values or when it sees that the PROPOSE from a new leader contains a STATUS for a lower view from a node which had cast VOTE2 for a relatively higher view in a previous round.

5.2 Detecting Safety Violation

Forensics deals with identifying the malicious nodes after a safety violation has happened. An interesting question is how to detect a violation itself without knowing which the honest nodes are. The definition of safety is that "two *honest* nodes must not commit different values". If the honest nodes were known beforehand, the fault tolerance would be trivial. This leads to a circular argument. In the original PBFT work, the client waits for $t + 1$ replies before deciding on the value to ensure that it gets at least one honest node to commit. To detect a violation, we can require the honest nodes to send their committed value to all other nodes and to the client. This way the nodes can monitor for a safety violation.

An interesting proposition is to combine safety violation detection with forensics. By delaying the actual commit by sometime until we are sure that a violation has not occurred, we can have a BFT protocol that can tolerate more than $n/3$ faults. The idea has been formalized as multi-threshold BFT here [2]. Note that adding a delay in commit is like making the network 'more synchronous'. In synchronous networks, BFT protocols can be designed to tolerate any number of faults (eg: Dolev-Strong). Thus the ability to tolerate more than $n/3$ faults by adding a delay to commit is not very surprising. However, we gain in latency as we do not wait for $f + 1$ 'rounds' to ensure perfect safety as in synchronous protocols.

5.3 Liveness Violation

The paper looked at safety violation foresnics. But having more than t faulty nodes can also cause liveness violation. All the malicious nodes must do is keep quiet and not send any votes. With $f > t$, protocol can not end without support from the malicious nodes. Even if the liveness is relaxed to an all-or-nothing commit, malicious nodes can still cause violation by supporting only few honest nodes to commit. One could detect such liveness violation by monitoring the view number changes. If the view number keeps on increasing without a consensus, it is likely that the malicious nodes are holding up consensus (or that there is a network partition). However, an irrefutable proof can not be provided this way.

5.4 Actual Communication cost

The paper defines k as the minimum number of transcripts required to prove culpability. But this does not reflect the actual communication cost involved in coming up with that proof. In case of violation across views as in 4.1.2, it is important to find e^*. Doing so will involve searching through the messages logs of all honest nodes for all views between e and e'. Thus, even if the final proof requires only one transcript (plus the commit message), the actual communication cost is higher as the client must talk to all nodes and there is a computational cost of searching through the transcripts.

5.5 Does having some trusted nodes help?

One of the reasons for impossibility of forensics is inability to provide an 'irrefutable proof', as in the case of PBFT-MAC. In a BFT protocol, it can happen that an honest node identifies a malicious node from the messages it received, but may not be able to prove so because the messages were not signed. By adding some trusted nodes in the setup, it might be possible to have forensics in protocols where it is not possible generally. An interesting question is, given a protocol, how many trusted nodes do you need to ensure forensic support.

6 CONCLUSION

In this document, we discuss the notion of forensics for BFT algorithms-what happens after a safety violation has occurred due to a sufficiently large number of adversarial nodes. We analyze the landscape through the lens of PBFT-PK, show that it has the strongest possible support and describe the forensics algorithm that provides it. We also identify some attacks that can cause safety violations. Further, we identify some costs incurred by the forensics algorithm that are not discussed in [4]. We also provide some rudimentary analysis of what it means to detect a safety violation and if it is possible to integrate such detection into the actual run of the algorithm and identify integration of trusted nodes as future work.

REFERENCES

[1] Ittai Abraham. Accessed on 4/5/2022. Synchrony, Asynchrony and Partial synchrony. https://decentralizedthoughts.github.io/2019-06-01-2019-5-31-models/.
[2] Ling Ren Atsuki Momose. Accessed on 4/5/2022. Multi-threshold BFT. https://eprint.iacr.org/2021/671.pdf.

[3] Ling Ren. Accessed on 4/5/2022. Lecture 3: Consensus and Replication, CS 598 FTD. https://sites.google.com/view/cs598ftd/home.

[4] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. 2021. BFT protocol forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1722–1743.

7 APPENDIX

Algorithm 1: PBFT-PK: initial value v_i (notation modified from [4])

```
1  $LOCK \leftarrow (0, v_\perp, \sigma_\perp)$ with selectors $e, v, \sigma$    // 0, $v_\perp$, $\sigma_\perp$: default view, value, and signature
2  $e \leftarrow 1$
3  while true do
       // PRE-PREPARE and PREPARE
4      as a leader
5          collect ⟨STATUS, $e - 1$, .⟩ from $2t + 1$ distinct replicas as a status certificate M
6          $v \leftarrow$ locked value with the highest view number in $M$
7          if $v = v_\perp$ then
8              $v \leftarrow v_i$
9          end
10         broadcast ⟨PROPOSE, $e, v, M$⟩
11     as a replica
12         wait for valid ⟨PROPOSE, $e, v, M$⟩ from leader
13         send ⟨VOTE1, $e, v$⟩ to leader
       // COMMIT
14     as a leader
15         collect ⟨VOTE1, $e, v$⟩ from $2t + 1$ distinct replicas as the collection $\Sigma$
16         $\sigma \leftarrow$ aggregate-sign($\Sigma$)
17         broadcast ⟨COMMIT, $e, v, \sigma$⟩
18     as a replica
19         wait for ⟨COMMIT, $e, v, \sigma$⟩ from leader
20         $LOCK \leftarrow (e, v, \sigma)$
21         send ⟨VOTE2, $e, v$⟩ to leader
       // REPLY
22     as a leader
23         collect ⟨VOTE2, $e, v$⟩ from $2t + 1$ distinct replicas as the collection $\Sigma$
24         $\sigma \leftarrow$ aggregate-sign($\Sigma$)
25         broadcast ⟨REPLY, $e, v, \sigma$⟩
26     as a replica
27         wait for ⟨REPLY, $e, v, \sigma$⟩ from leader
28         output $v$ and send ⟨REPLY, $e, v, \sigma$⟩ to the client
29     call VIEWCHANGE()
30 end
31 if a replica encounters timeout in any "wait for", call procedure VIEWCHANGE()
32
33 procedure VIEWCHANGE()
34     broadcast ⟨BLAME, $e$⟩
35     collect ⟨BLAME, $e$⟩ from t + 1 distinct replicas, broadcast them
36     quit this view
37     send ⟨STATUS, $e, LOCK$⟩ to the next leader
38     enter the next view, $e \leftarrow e + 1$
39 function VALID(⟨PROPOSE, $e, v, M$⟩)
40     $v^* \leftarrow$ the locked value with the highest view number in $M$
41     if ($v^* = v$ or $v^* = v_\perp$) and ($M$ contains locks from $2t + 1$ distinct replicas) then
42         return true
43     else
44         return false
45     end
46 end
```

Algorithm 2: Forensics algorithm for PBFT-PK (notation modified from [4])

```
1  as a replica running PBFT-PK
2      $Q \leftarrow$ all PROPOSE messages in transcript
3      upon receiving $\langle$REQUEST-PROOF, $e, v, e'\rangle$ from a client
4          for $m \in Q$ do
5              $(v'', e'') \leftarrow$ the highest lock in $m.M$
6              if $m.e \in (e, e']$ and $v'' \neq v$ and $e'' \leq e$ then
7                  send $\langle$PROPOSE, $m\rangle$ to the client
8              end
9          end
10     end
11 as a client
12     upon receiving two conflicting REPLY messages
13         if the two messages are from different views then
14             $\langle$REPLY, $e, v, \sigma\rangle \leftarrow$ the message from lower view
15             $e' \leftarrow$ the view number of REPLY from higher view
16             broadcast $\langle$REQUEST-PROOF, $e, v, e'\rangle$
17             wait for: $\langle$PROPOSE, $m\rangle$ s.t $m.e \in (e, e']$ and $v'' \neq v$ and $e'' \leq e$ where $(v'', e'')$ is the highest lock in $m.M$
18             if in $m.M$ there are two locks $(e'', v_1, \sigma_1)$ and $(e'', v_2, \sigma_2)$ s.t $v_1 \neq v_2$ then
19                 output $\sigma_1 \cap \sigma_2$
20             else
21                 output the intersection of senders in $m.M$ and signers of $\sigma$
22             end
23         else
24             $\langle$REPLY, $e, v, \sigma\rangle \leftarrow$ first REPLY message
25             $\langle$REPLY, $e, v', \sigma'\rangle \leftarrow$ second REPLY message
26             output $\sigma \cap \sigma'$
27         end
28     end
```