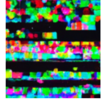


ZKSECURITY

Audit of Aleo's consensus

October 30th, 2023

Introduction

On October 9th, 2023, zkSecurity was tasked to audit Aleo's consensus for use in the **Aleo** blockchain. Two consultants worked over the next 3 weeks to review Aleo's codebase for security issues.

The code was found to be thoroughly documented and of high quality. In addition, the team acted in a highly cooperative way and was key in helping us find a number of the issues in this report.

Below, we go through the scope and a few general recommendations. The rest of this report includes an overview of the protocol as well as the findings.

Scope

zkSecurity used the `testnet3-audit-zks` branch (commit `9234cbee73666c7e6d00dfbdbcb947a244a43818`) of the **snarkOS** repository.

Included in scope was:

- Aleo's **Bullshark** (the partially synchronous version) and **Narwhal** implementation for resharing and coming to consensus on a global order of transactions.
- Aleo's high-level consensus logic in snarkOS (within some folders of snarkOS/node)
- Aleo's ledger service, that has dependencies living in the **snarkVM** repository.

Note that some logic (like the puzzle mechanism) was out of scope, as it was planned for replacement.

General Recommendations

In addition to addressing the findings listed in this report, we offer the following strategic recommendations to Aleo:

Specify the protocol implemented. Bullshark, as it was introduced in two white papers, is loosely specified. Furthermore, the implementation of Aleo represents an important departure from the protocol described in the papers: the commit rule (discussed in finding **Commit Flow Can Lead To Safety Violation**), the garbage collection (discussed in finding **Garbage Collection Can Block Commits From Happening**), and the dynamic committee feature (discussed in finding **Dynamic Committee Feature is Not Safe**), all deviate from the whitepaper version of Bullshark. This makes it hard to understand the protocol, and to reason about its security. We strongly recommend that Aleo writes a specification of the protocol implemented.

Create test vectors. It might benefit Aleo to implement an embedded-DSL, as well as test vectors, to test a suite of interesting scenarios. For example, a number of scenarios like the one described in **Commit Flow Can Lead To Safety**

Violation could be useful to heuristically check that properties from the Bullshark protocol are well-understood and preserved in the implementation.

Bullshark in Aleo

In this section we give an overview of the consensus protocol implemented by Aleo, namely Bullshark.

The Bullshark consensus protocol is the evolution of a long lineage of consensus protocol, perhaps starting with the very first practical one (PBFT). The idea remains more or less the same in most (all?) of these protocols: the set of participant is fixed, and they advance together in rounds:

1. In the first round, a "leader" broadcasts a proposal.
2. In the second round, participants vote on the proposal.
3. In the third round, participants vote on votes.

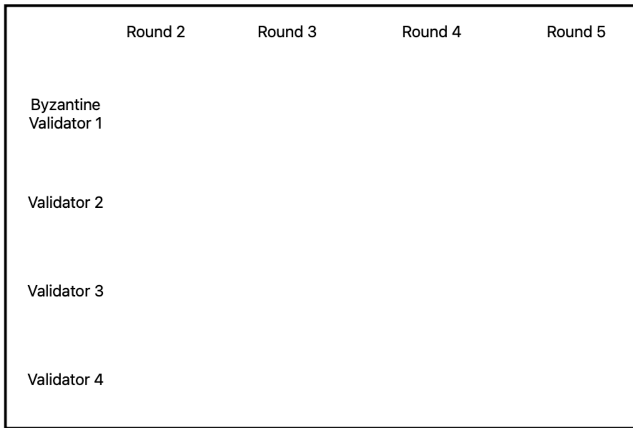
If you see enough participants who have voted on a proposal, and enough participants have seen that (the "vote on votes" part), then we can "commit" to a proposal. In blockchain lingo, committing a proposal would be to finalize and apply the transactions it contains to the blockchain's state.

This basic 3-step committing process was never really improved. Instead, most major optimizations have been obtained by using pipelining techniques. For example, The **Hotstuff** protocol introduced pipelining of this flow by having a leader propose in every round. More recently, Bullshark introduced another type of pipelining by having every participant propose in every round. (Another recent protocol **Shoal** introduces both pipelining techniques in a single protocol.)

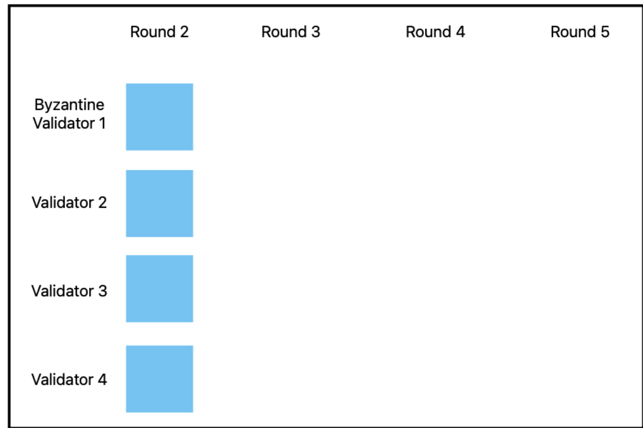
More concretely, a round of bullshark between $3f + 1$ participants (of which only f can be malicious) goes like this:

1. At the start of a round, every participant broadcasts a new batch of transactions (which can be thought of as a block) to every other participant. (See at the end of the list how the batch was created.)
2. If you receive a batch from someone, make sure that it's the only batch you've seen from them in this round, store it, and sign it.
3. Once you collect $2f + 1$ signatures on your batch, this produces a certificate for your batch, which you can broadcast to everyone.
4. Once you observe $2f + 1$ certificates in this round, produce a batch of transactions for the next round in which you also include the $2f + 1$ observed certificates. This should look like a directed acyclic graph (or DAG) of batches.
5. Go back to step 1.

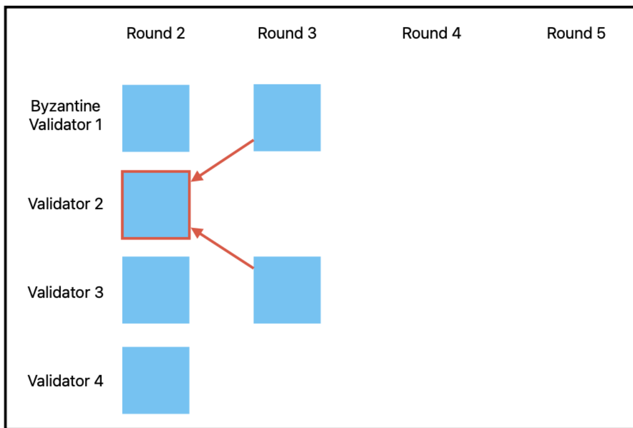
To explain concrete examples in the rest of this report, we use the following types of diagrams:



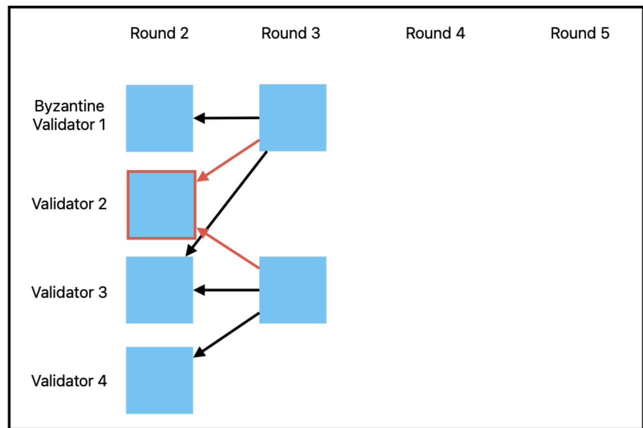
Simple examples can be given with 4 validators, and a single byzantine validator (f=1)



In these diagrams, the blue boxes are certified batches, which means that they've obtained 2f+1 signatures from validators



As such, an anchor (red borders) is committed if it is linked by f+1=2 certified proposals



In addition, each proposal must link to 2f+1 = 3 previous proposals

This still doesn't let us know how participants agree on what gets committed. The "commit rule" of Bullshark is quite simple: in every even round (e.g. round 2, 4, etc.) a leader is chosen (for example, via a round-robin election), and their DAG (their batch and all the batches they link to transitively) gets committed if there's $f + 1$ certificates referring to it.

A leader's batch is called an **anchor**, and it is possible that an anchor does not get committed in an even round. But if an anchor gets committed by some honest nodes, it will always be committed later on by all other honest nodes. This is because a committed anchor will always be included in the DAG of another anchor.

This property is called the **quorum intersection** property and can be proven quite easily. If an anchor is committed in an even round i, it is because it was referred to by $f + 1$ certificates in round $i + 1$. Let's call this set A. Any block (including anchor block) in round $i + 2$ will have to include $2f + 1$ certificates from round $i + 1$. let's call that set B.

We need to show that $A \cap B \neq \emptyset$, or in other words that the intersection contains at least one certificate, or in other words that when a block is committed then there'll always be a path leading to that block in any block from the next even round. This is not too hard to show: the multiset $|A \cup B| = (f + 1) + (2f + 1) = 3f + 2$ which is 1 vertex more than the total number of vertices in a round and thus has some multiplicities.

As leaders and their anchors can be slow to be proposed and certified, timeouts are added to ensure that participants will wait a certain amount of time for the leader to show up before moving to the next round. Concretely, in even rounds, participants will wait to see an anchor to certify, and will wait for a certified anchor to include in their next

proposal. In odd rounds, participants will wait to see votes for an anchor to certify, and will wait for certified votes for an anchor to include in their next proposal.

One last (important) subtlety is that as described, the protocol allows for a node's storage to grow ad-infinitum if they do not observe any commit. To prevent this to happen, a **garbage collection** protocol is added in order to prune older parts of the DAG as it grows. As such, when a commit happens, the DAG will be traversed up to some "garbage collection frontier", and everything on the left of that frontier will be discarded.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	snarkOS/node/narwhal/bft	Commit Flow Can Lead To Safety Violation	High
01	snarkOS/node/narwhal/primary	Lack of Dag Containment Could Lead To Safety Violation	High
02	snarkOS/node/narwhal	Dynamic Committee Feature is Not Safe	High
03	snarkOS/node/narwhal	Liveness Issue Due To Intolerance To Malleable Certificates	High
04	snarkOS/node/narwhal	Garbage Collection Can Block Commits From Happening	Medium
06	snarkOS/node/narwhal	Byzantine Behavior is Not Detected	Low
08	snarkVM/ledger/committee	Potentially Biased Leader Election	Informational
09	snarkVM/ledger	Redundant Serialization Could Lead To Manipulating Unsanitized Inputs	Informational

00 - Commit Flow Can Lead To Safety Violation

- **snarkOS/node/narwhal/bft**

High

Description. As of now, the Bullshark commit flow can lead to situations where different participants have a different commit history.

Let's look at the following scenario (pictured in the diagram below): An anchor A_1 at round 2 gets committed. This means that in round 3 there are $f + 1$ certified votes that are produced. In other words, in round 3, $f + 1$ batches that include A_1 in their $2f + 1$ edges are certified.

In Bullshark, a commit leads a validator to deterministically traverse the DAG pointed by the anchor, and to restart the committing process at every anchor encountered. Yet, in Aleo, the entire DAG --including all previous non-committed anchors-- is committed as a single block. As we will see, this is the source of the problem.

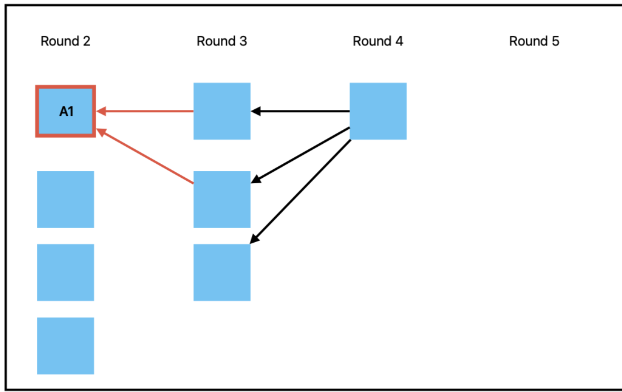
Next, imagine that the leader (V_1) is byzantine. Thus, they decide to not release the certificate they produced in round 3. The other validators will move to the next round without V_1, and nobody but the byzantine leader will commit A_1.

In round 4, another validator (V_4 in the diagram) produces an anchor A_2, which then gets committed (as it gets $f + 1$ certified votes in round 5).

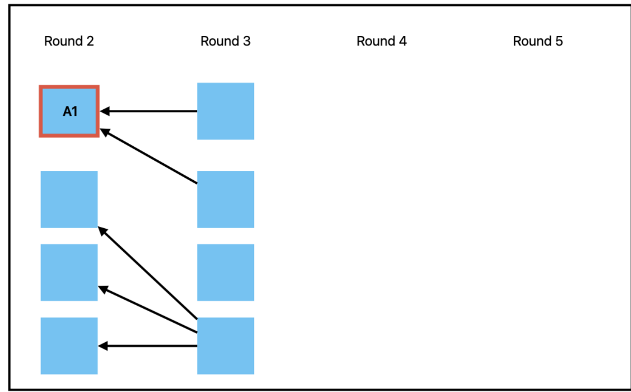
At this point, validators will commit to A_2 as soon as they see $f + 1$ certified votes in round 5. Before this happens, the byzantine validator V_1 can send their certified vote from round 3 (or round 4) to a select group of validators to get them to commit A_1 as a block at height i before they get to commit A_2. This group will then commit A_2 as an additional block at height $i + 1$.

The other group $(V_4$ in the diagram) will commit A_2 and A_1 as a single block at height i.

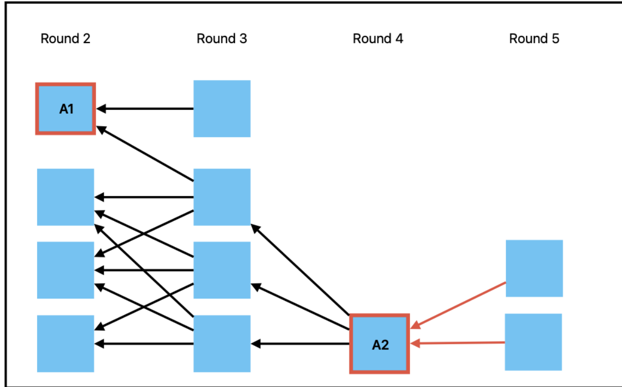
> *"Note: due to the dynamic committee, this scenario is slightly different, what really happens is that the first group will most likely produce a different committee after the first commit, which might lead to a different leader being elected in round 4 and thus the block not being committed."*



V1 sees the f+1 commits so commits A1

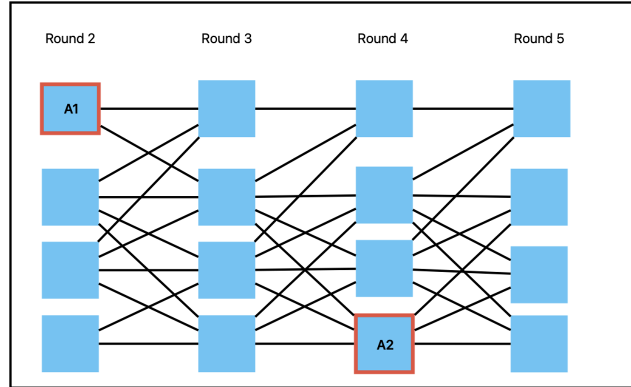


V4 includes other blocks so doesn't see the commit



V4 ends up seeing a commit on A2, so commits to A2 and A1 as a single block

At this point V1 might also see the commit of A2 and so Commit to a DAG up until A1 as a new block



Full example

There are two important subtleties to understand the attack here:

1. The quorum intersection property of Bullshark does not ensure that all participants will always see anchors being committed in the same order. It only ensures that if an anchor is committed, then any future anchor will have a path to it in its DAG.
2. It is possible to form a certificate that doesn't get included in validators' DAG, either because you are too slow to broadcast it (and validators will move on to the next round including other certificates) or because you are byzantine and choose to broadcast your certificate at a later time.

To emphasis, the previous attack works because:

- The last validator V_4 never sees the certificates from V_1.
- In round 3, the byzantine validator V_1 obtains a certificate for their batch but doesn't get it included by other validators
- In round 4 or 5 validators V_2 and V_3 see the certificate of V1 in round 3 which leads them to commit A_1.

Recommendations. Follow the commit logic of the Bullshark paper by restarting the commit process at every anchor, when traversing a DAG that is to be committed.

01 - Lack of Dag Containment Could Lead To Safety Violation

- **snarkOS/node/narwhal/primary**

High

Description. A BFT protocol's resilience to forking (having multiple honest participants share contradicting views) comes from its **safety** property. In turn, the safety proof of a BFT protocol relies on several properties and assumptions. For example, Bullshark relies on the assumption that a participant in its protocol cannot obtain two different certified batches of transactions in the same round.

In the **Narwhal** paper (which Bullshark builds on top of), this property is called **containment**:

> *"Lemma A.5. The DAG protocol satisfies Containment."*

The same paper includes the following containment proof, which includes (with our emphasize on) the rules that are important to encode in a node implementation:

> *"Proof. Every block contains its author and round number and **honest validators do not sign two different blocks in the same round from the same author**. Therefore, since $2f + 1$ signatures are required to certify a block, **two blocks in the same round from the same author can never be certified**. Thus, for every certified block that honest validators locally store, they always agree on the set of digest in the block (references to blocks from the previous round). The lemma follows by recursively applying the above argument starting from the block b'."*

Later on, it is used in the safety proof:

> *"Lemma 2. Any two honest validators commit the same sequence of block leaders. Since after ordering a block leader each validator orders the leader's causal history by some pre-define deterministic rule, **we get that by the Containment property of Narwhal all honest validators agree on the total order of the DAG's blocks.**"*

The current node implementation we looked at does not have a participant store batch proposals (also called collections in other implementations). Instead, a participant who receives a batch will store the batch's transmissions/transactions. As such, a participant will happily sign different batches from the same author in the same round, violating the containment property.

In practice, this means that a malicious participant could create two certificates for two contradicting blocks at the same round.

Note that we could not find a way to exploit this to fork the protocol, due to the fact that a byzantine validator cannot get $f + 1$ certified votes on two conflicting proposals. This is because none of the two partitions of validators (of size $f + 1$) will be able to certify their proposals (containing one of the byzantine proposals) as this will contradict the view of the other partition that needs to sign their proposal.

None-the-less, this would lead to a liveness issue similar to the one described in **Liveness Issue Due To Intolerance To Malleable Certificates**. Depending on how the latter issue is fixed, or how other subtleties could come into play, we deemed that this finding might lead to safety issues if not fixed.

Reproduction steps. One can observe that participants happily sign different batches from the same author in the same round by adding the following code to the `Primary::propose_batch` function:

```
         // Generate the local timestamp for batch
         let timestamp = now();
         // Prepare the transmission IDs.
-        let transmission_ids = transmissions.keys().copied().collect();
+        let transmission_ids: indexmap::IndexSet<TransmissionID<N>> =
transmissions.keys().copied().collect();
         // Prepare the certificate IDs.
-        let certificate_ids = previous_certificates.into_iter().map(|c|
c.certificate_id()).collect();
+        let certificate_ids: indexmap::IndexSet<Field<N>> =
+            previous_certificates.into_iter().map(|c| c.certificate_id()).collect();
         // Sign the batch header.
-        let batch_header = BatchHeader::new(private_key, round, timestamp, transmission_ids,
certificate_ids, rng)?;
+        let batch_header =
+            BatchHeader::new(private_key, round, timestamp, transmission_ids.clone(),
certificate_ids.clone(), rng)?;
+
         // Construct the proposal.
-        let proposal =
-            Proposal::new(self.ledger.get_previous_committee_for_round(round)?,
batch_header.clone(), transmissions)?;
+        let proposal = Proposal::new(
+            self.ledger.get_previous_committee_for_round(round)?,
+            batch_header.clone(),
+            transmissions.clone(),
+        )?;
+        println!(
+            "ZKSEC: proposed batch id {} in round {} with {} transactions",
+            batch_header.batch_id(),
+            batch_header.round(),
+            transmission_ids.len()
+        );
         // Broadcast the batch to all validators for signing.
         self.gateway.broadcast(Event::BatchPropose(batch_header.into()));
+
+        // ZKSEC: create another one for fun!
+        if self.evil {
+            let mut transmissions2 = transmissions.clone();
+            transmissions2.pop();
+            let transmission_ids2: indexmap::IndexSet<TransmissionID<N>> =
transmissions2.keys().copied().collect();
```

```
+           let batch_header2 =
+               BatchHeader::new(private_key, round, timestamp, transmission_ids2.clone(),
certificate_ids, rng)?;
+
+           println!(
+               "ZKSEC: _also_ proposed batch id {} in round {} with {} transactions",
+               batch_header2.batch_id(),
+               batch_header2.round(),
+               transmission_ids2.len()
+           );
+           self.gateway.broadcast(Event::BatchPropose(batch_header2.into()));
+       }
+
```

and the following code in `` `Primary::process_batch_signature_from_peer` ``:

```
        // Retrieve the signature and timestamp.
        let BatchSignature { batch_id, signature, timestamp } = batch_signature;

+        // ZKSEC
+        if true {
+            println!(
+                "ZKSEC: received signature from {} for batch id {}",
+                signature.compute_key().to_address(),
+                batch_id,
+            )
+        }
+
```

Running an end-to-end test like `` `test_state_coherence` `` (`` `cargo test --package snarkos-node-narwhal --test bft_e2e -- test_state_coherence --exact --nocapture --ignored` ``) can expose that participants will happily sign two different batches in the same round:

```
ZKSEC: proposed batch id
6424465404037277811535918421886783076581165244697445935535915255730835352828field in round 20
with 250 transactions
ZKSEC: _also_ proposed batch id
2130383929461709954064380322325477721698409715003851135305445666084395401543field in round 20
with 249 transactions
// ...
ZKSEC: received signature from aleo18fsar6muz3ksa68gz0qp5vkxm7vh07f7pctmkzxwlf9adxzhw59qwy7fwm
for batch id 6424465404037277811535918421886783076581165244697445935535915255730835352828field
// ...
ZKSEC: received signature from aleo18fsar6muz3ksa68gz0qp5vkxm7vh07f7pctmkzxwlf9adxzhw59qwy7fwm
for batch id 2130383929461709954064380322325477721698409715003851135305445666084395401543field
```

Recommendations. To help avoid signing two batches from the same validator in the same round, have nodes store a hashmap of authors to `` `(round, batch_id)` `` tuples, in addition to storing batches' transactions.

02 - Dynamic Committee Feature is Not Safe

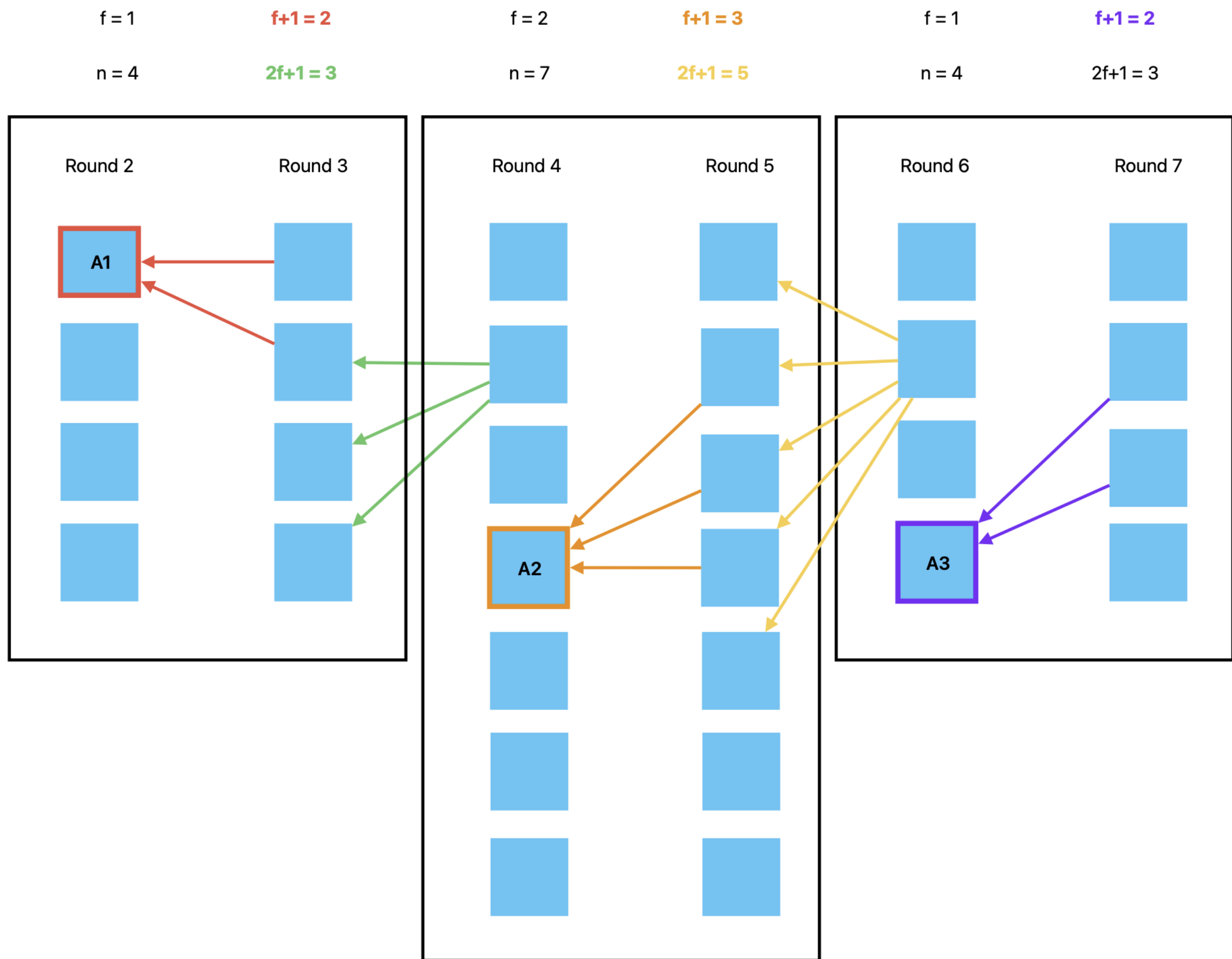
- snarkOS/node/narwhal

High

Description. Aleo's implementation of Bullshark includes a new feature which allows the validator set to update itself on every new block created. Where a block is an object containing a DAG of batches of transactions that are being committed, and potentially including several anchor batches (as explained in **Commit Flow Can Lead To Safety Violation"**).

This dynamic committee feature is not included in the Bullshark or Narwhal papers, and is not specified by Aleo, which makes it hard to understand if the protocol is safely designed and implemented. Potential issues could exist where the liveness of the network is impacted by nodes being stuck by not being able to understand what the current set of validators is, or worse safety issues could exist where the dynamic change of the quorum threshold required to commit leads to forks.

Currently, the reconfiguration works by allowing a change of committee at every single block. This committee of validators is dictated by the execution of smart contracts triggered by the transactions of a committed block. A different committee means that different validators, with different stakes, will now be part of advancing the consensus protocol. We summarize how reconfiguration affects the protocol in the following diagram:



Reconfiguring a validator set is arguably one of the most tricky features to get right. In the rest of this section we give a number of examples of unexpected or surprising behavior that comes from the addition of a dynamic committee.

Handling outdated quorum size. As the set of validators as well as their stake can change dynamically, and validators might have missed commits, the value of $f + 1$ (resp. $2f + 1$) to commit (resp. certify) vertices might appear too low or too high to a validator.

This can result in a scenario where a validator could wrongly commit to a batch: a validator could believe that a certified batch is both an anchor and has received enough certified votes due to being on an outdated committee.

While this issue seems addressable by fixing **Commit Flow Can Lead To Safety Violation"**, it still can lead to previous non-committed batches being committed. For example, if the previous scenario leads to triggering a commit, which leads to traversing a DAG to a previous anchor, that anchor will be committed (due to being in the path) even if it should not have been committed.

Handling new validators. Could a node be stuck because they can't get enough messages to advance in the protocol, due to the new validators not being able to communicate with the lagging node? As can be seen in `` `process_batch_certificate_from_peer` ``, new validators messages are discarded:

```
async fn process_batch_certificate_from_peer(
```

```
        &self,
        peer_ip: SocketAddr,
        certificate: BatchCertificate<N>,
    ) -> Result<()> {
        // TRUNCATED...
        if !self.gateway.is_authorized_validator_address(author) {
            // Proceed to disconnect the validator.
            self.gateway.disconnect(peer_ip);
            bail!("Malicious peer - Received a batch certificate from a non-committee member
({author})");
```

where `is_authorized_validator_address` can only consider the previous and current committee:

```
    /// Returns `true` if the given address is an authorized validator.
    pub fn is_authorized_validator_address(&self, validator_address: Address<N>) -> bool {
        // Determine if the validator address is a member of the previous or current committee.
        // We allow leniency in this validation check in order to accommodate these two
scenarios:
        //  1. New validators should be able to connect immediately once bonded as a committee
member.
        //  2. Existing validators must remain connected until they are no longer bonded as a
committee member.
        //     (i.e. meaning they must stay online until the next block has been produced)
        self.ledger
            .get_previous_committee_for_round(self.ledger.latest_round())
            .map_or(false, |committee| committee.is_committee_member(validator_address))
            || self
                .ledger
                .current_committee()
                .map_or(false, |committee| committee.is_committee_member(validator_address))
    }
```

Handling timers. Timeout-related logic designed in non-dynamic settings might not be working as intended in dynamic settings. This might affect the liveness of the protocol. For example, lagging validators will wait and attempt to include the wrong certificates in their edges in odd rounds:

```
// Compute the stake for the leader certificate.
let (stake_with_leader, stake_without_leader) =
    self.compute_stake_for_leader_certificate(leader_certificate_id, current_certificates,
&previous_committee);
// Return 'true' if any of the following conditions hold:
stake_with_leader >= previous_committee.availability_threshold()
    || stake_without_leader >= previous_committee.quorum_threshold()
    || self.is_timer_expired()
```

as well as in even rounds:

```
// Determine the leader of the current round.
```

```rust
let leader = match previous_committee.get_leader(current_round) {
    Ok(leader) => leader,
    Err(e) => {
        error!("BFT failed to compute the leader for the even round {current_round} - {e}");
        return false;
    }
};
// Find and set the leader certificate, if the leader was present in the current even round.
let leader_certificate = current_certificates.iter().find(|certificate| certificate.author() ==
leader);
*self.leader_certificate.write() = leader_certificate.cloned();

self.is_even_round_ready_for_next_round(current_certificates, previous_committee, current_round)
```

Nodes will also perform checks on incorrect thresholds:

```rust
if self.is_timer_expired() {
    debug!("BFT (timer expired) - Checking for quorum threshold (without the leader)");
    // Retrieve the certificate authors.
    let authors = certificates.into_iter().map(|c| c.author()).collect();
    // Determine if the quorum threshold is reached.
    return committee.is_quorum_threshold_reached(&authors);
}
```

Handling new connections. In the layers below, connecting to new validators (or ignoring old ones) might lead to unexpected behavior. For example, in `` `Primary::propose_batch` ``, the primary will not propose a batch if they are not connected to enough validators (according to their own view of the validator set):

```rust
// Check if the primary is connected to enough validators to reach quorum threshold.
{
    // Retrieve the committee to check against.
    let committee = self.ledger.get_previous_committee_for_round(round)?;
    // Retrieve the connected validator addresses.
    let mut connected_validators = self.gateway.connected_addresses();
    // Append the primary to the set.
    connected_validators.insert(self.gateway.account().address());
    // If quorum threshold is not reached, return early.
    if !committee.is_quorum_threshold_reached(&connected_validators) {
        debug!(
            "Primary is safely skipping a batch proposal {}",
            "(please connect to more validators)".dimmed()
        );
        trace!("Primary is connected to {} validators", connected_validators.len() - 1);
        return Ok(());
    }
}
```

Recommendations. This finding is a tricky one as it appears that fixing the dynamic committee feature is not trivial. Ensuring that safety is correctly guarded throughout validator set changes, especially as validators might have an outdated view of the committee while advancing through newer rounds of the protocol, is not easy.

We recommend specifying a solution and writing up a proof that it is safe. Note also that recently Sui Lutris came out with a description of committee reconfigurations in a similar protocol.

03 - Liveness Issue Due To Intolerance To Malleable Certificates

- **snarkOS/node/narwhal**

High

Description. There's a few occasions in which a validator can observe other validators' batch certificates. For example, validators send certificates they create to each other during a round, or at the beginning of the round they also send each other batches that contain $2f + 1$ batch certificates from the previous round.

When that happens, a validator moves on to processing each batch certificate it sees. If it's a new certificate that it hasn't seen, and if it's the first certificate that it sees for this author and at this round, then it will pass it to the BFT module.

But before that, it will ensure that it has enough information to construct the directed acyclic graph (DAG) that the batch certificate points to. It does that by recursively traversing the DAG and collecting each batch certificate that is missing.

The problem is that participants in Bullshark can easily create different certificates for the same batch. This is because a certificate is determined by its signatures, and a participant can include a different set of $2f + 1$ signatures to form a different certificate for a batch.

So it is possible that a validator has a specific batch certificate for a specific round in their storage, but needs to fetch what looks like a different batch certificate for the same batch. Unfortunately, the current implementation does not allow storing different `` `(round, author, certificate_id)` `` and `` `(round, author, certificate_id')` `` tuples where `` `certificate_id != certificate_id'` ``:

```
    pub fn check_certificate(
        &self,
        certificate: &BatchCertificate<N>,
        transmissions: HashMap<TransmissionID<N>, Transmission<N>>,
    ) -> Result<HashMap<TransmissionID<N>, Transmission<N>>> {
        // TRUNCATED...
        if self.contains_certificate_in_round_from(round, certificate.author()) {
            bail!("Certificate with this author for round {round} already exists in storage
{gc_log}")
        }
```

This leads to a liveness issue that can be exploited by a single byzantine participant:

1. Alice, a byzantine leader in round i, collects $2f + 2$ signatures which allows her to create two different certificates of $2f + 1$ signatures: cert_1 and cert_2.
2. Alice shares cert_1 with $f + 1$ participants, and cert_2 with $f + 1$ other participants.

3. These participants will continue to wait to receive certificates in round i. They will move to round $i+1$ once they manage to collect $2f+1$ certificates (including one of Alice's certificates).
4. In round $i+1$, validators will broadcast their proposals/batches and attempt to collect $2f+1$ signatures on them.
5. They will fail to do so as at most $n-(f+1)=(3f+1)-(f+1)=2f$ nodes will be able to validate their batch, which is not enough (a certificate requires $2f+1$ signatures).
6. The network will be stuck indefinitely.

Note that if the implementation allowed storing different certificates for the same batch, then a graver safety issue could happen. To understand why, continue reading.

Once the missing batch certificates have been fetched, and the BFT module has been involved, the `update_dag` function is called. That function reconstructs the DAG that the batch certificate points to and then tries to commit it.

When committing a DAG, a validator stops at the garbage collection frontier (where everything on the left of that frontier has been pruned and cannot be committed) as well as at anchors that have already been committed.

The DAG to commit is constructed using the helper function `order_dag_with_dfs`:

```
    /// Returns the subdag of batch certificates to commit.
    fn order_dag_with_dfs<const ALLOW_LEDGER_ACCESS: bool>(
        &self,
        leader_certificate: BatchCertificate<N>,
    ) -> Result<BTreeMap<u64, IndexSet<BatchCertificate<N>>>> {
        // TRUNCATED...
        while let Some(certificate) = buffer.pop() {
            // TRUNCATED...
            for previous_certificate_id in certificate.previous_certificate_ids().iter().rev() {
                // TRUNCATED...
                // If the previous certificate was recently committed, continue.
                if self.dag.read().is_recently_committed(previous_round,
*previous_certificate_id) {
                    continue;
                }
```

As one can see, it uses the notion of a certificate ID to detect committed certificates.

The problem is that a certificate ID is computed using the signatures it contains:

```
pub fn compute_certificate_id(batch_id: Field<N>, signatures: &IndexMap<Signature<N>, i64>) ->
Result<Field<N>> {
        let mut preimage = Vec::new();
        // Insert the batch ID.
        batch_id.write_le(&mut preimage)?;
        // Insert the signatures.
        for (signature, timestamp) in signatures {
            // Insert the signature.
```

```
            signature.write_le(&mut preimage)?;
            // Insert the timestamp.
            timestamp.write_le(&mut preimage)?;
        }
        // Hash the preimage.
        N::hash_bhp1024(&preimage.to_bits_le())
    }
```

Since a byzantine validator can create different valid certificates for an anchor they propose (as we've seen in the previously discussed liveness issue), it is possible that a validator ends up reconstructing parts of a DAG (from a batch certificate) that looks slightly different from a DAG that it has already committed before.

Traversing parts of that DAG, the committing function might not properly stop at batches that it had already committed, potentially recommitting entire parts of a DAG.

Recommendations. As certificates are naturally malleable, one could fix this issue by making sure to always interpret DAGs without their certificates, and only using certificates to ensure that a batch is usable in the BFT layer.

To summarize, the issue comes from the fact that the protocol fails to understand that two batches can be the same, in spite of having different certificates. As such encoding these equalities correctly should fix the issue.

04 - Garbage Collection Can Block Commits From Happening

- snarkOS/node/narwhal

Medium

Description. Without the concept of garbage collection, nodes in Bullshark might grow their local DAG ad-infinitum if no commit happens. Due to this, garbage collection (GC) is added to upperbound the memory requirement of a node to realistic numbers.

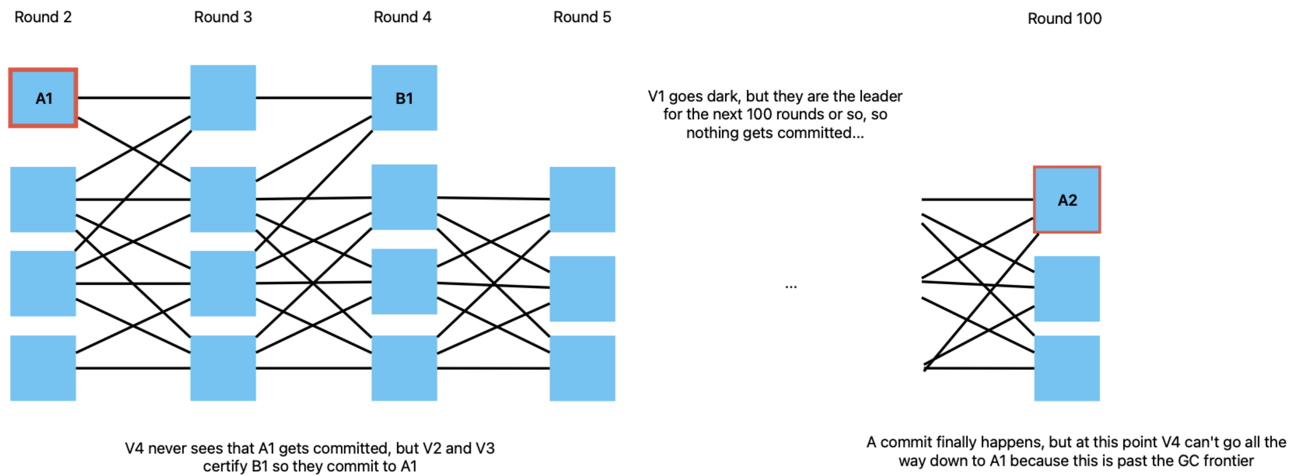
Commits are missed either because there's no proposals made by the leader in commit rounds (the even rounds), or because they are too slow to get it certified, or because of other similar issues (see **Commit Flow Can Lead To Safety Violation** for another example).

The GC logic of Aleo's Bullshark implementation is relevant in two places:

- When a validator advances in the protocol: Whenever the round of a validator is incremented (due to the natural flow of the protocol), an internal "GC round" is incremented. This GC round is implemented as exactly 50 rounds below the current round, and incrementing it triggers pruning of old data (certificates and potentially dangling transactions).
- When a validator commits a batch: A validator commiting a batch will traverse the DAG pointed at by the batch. During the traversal, they will ensure that they never go further than anchors they've already committed, or further than a "GC frontier" which is defined as 50 rounds below the highest committed round so far.

One problem with the previous GC rules is that what is being pruned in the first part is not necessarily content that will never be committed in the second part.

For example, an anchor A_1 in round 2 could get committed by some validators but not all, and if no further commit happens for more than 50 rounds then some of these validators won't be able to commit A_1 due to having pruned all information (transactions and certificates) related to A_1 and needed to commit the anchor.



Recommendations. The Bullshark whitepaper uses timestamps instead of a fixed number of rounds to define what can be garbage collected:

> *"Since by the properties of the underling reliable broadcast all parties agree on the causal histories of the leaders, once parties agree which leaders to order they also agree what rounds to garbage collect."*

This approach might be able to prevent validators from garbage collecting batches that will need to be committed in the future.

06 - Byzantine Behavior is Not Detected

- **snarkOS/node/narwhal**

Low

Description. Bullshark works based on a number of assumptions, including that its participants comprise a majority of honest nodes (or non-byzantine nodes).

Such nodes must follow some rules to the letter. As long as these rules are followed, and the threshold of tolerable byzantine nodes is not exceeded, the protocol is safe (no forks happen) and live (transactions continue to be committed).

That being said, the protocol should attempt to detect when nodes are not following these rules (to potentially penalize them or investigate bugs) to the extent of what's acceptable and possible.

For example, when receiving a certificate from a different peer, a validator will silently replace any previous certificate seen from this peer at the same round, even if different:

```
/// Inserts a certificate into the DAG.
pub fn insert(&mut self, certificate: BatchCertificate<N>) {
    let round = certificate.round();
    let author = certificate.author();

    // If the certificate was not recently committed, insert it into the DAG.
    if !self.is_recently_committed(round, certificate.certificate_id()) {
        // Insert the certificate into the DAG.
        let _previous = self.graph.entry(round).or_default().insert(author, certificate);
```

More seriously, and as described in <u>Lack of Dag Containment Could Lead To Safety Violation</u>, a validator will happily handle different proposals at the same round from the same author.

Recommendations. We recommend logging any such instances as at worst they can provide insight as to who is acting maliciously, and at best they can lead to detecting bugs in the Bullshark implementation early.

08 - Potentially Biased Leader Election

- **snarkVM/ledger/committee**

Informational

Description. To choose the leader for a round, given a current committee, the hash (of some metadata) modulo the total stake is taken as a needle to point to the leader's stake.

In pseudo-code the logic looks like that:

```
seed = [starting_round, current_round, total_stake]
digest = hash_to_group_psd4(seed).x_coordinate()
stake_index = digest % total_stake
current_stake_index = 0
for candidate, stake in candidates:
    current_stake_index += stake
    if stake_index < current_stake_index:
        leader = candidate
        break
return leader
```

The `` `digest % total_stake` `` operation, if required to be fair, needs to uniformly span the range of numbers from 0 to `` `total_stake` ``.

This is of course not always the case as `` `total_stake` `` is different depending on the committee. As such some members of a committee will have more (or less) chance to be elected than others.

This might be an acceptable bias, especially as the Bullshark protocol provides a property called "chain quality". From the Narwhal paper:

> *"A fourth step provides Chain Quality by imposing restrictions on block creation rate. Each block from a validator contains a round number, and must include a quorum of certificates from the previous round to be valid. As a result, a fraction of honest validators' blocks are included in any proposal. Additionally, a validator cannot advance to a Mempool round before some honest ones concluded the previous round, preventing flooding. As a result Narwhal provides the consensus layer censroship-resistence (as defined in HoneyBadger BFT) without the need for using any additional mechanisms such as threshold encryption."*

That being said, leader bias might have other issues such as a leader intentionally targeting commits that will get them re-elected, which in turns might get them to collect more coinbase rewards or slow down the protocol.

09 - Redundant Serialization Could Lead To Manipulating Unsanitized Inputs

- snarkVM/ledger

Informational

Description. Serialization (and deserialization) of data structures is done manually for every different type that can be exchanged through the network.

Structures that are being serialized often contain fields that represent cached values that can be derived from the structure itself. While these fields do not necessarily need to be serialized, as they can be recomputed, they are almost always serialized. This means that deserialization must be done carefully to ensure that any of these cached values actually contain the correct value.

For example, a `BatchHeader` structure contains a `batch_id` field which represents a hash of its content. Thus, deserialization from bytes will recompute that value and check that it matches the deserialized value:

```
impl<N: Network> FromBytes for BatchHeader<N> {
    /// Reads the batch header from the buffer.
    fn read_le<R: Read>(mut reader: R) -> IoResult<Self> {
        // TRUNCATED..

        // Read the batch ID.
        let batch_id = Field::read_le(&mut reader)?;

        // TRUNCATED...

        // Construct the batch.
        let batch = Self::from(author, round, timestamp, transmission_ids,
previous_certificate_ids, signature)
            .map_err(|e| error(e.to_string()))?;

        // Return the batch.
        match batch.batch_id == batch_id {
            true => Ok(batch),
            false => Err(error("Invalid batch ID")),
        }
```

This pattern is error-prone as if someone forgets to check these fields, they could contain incorrect values, which in turn could lead to unexpected behavior and potentially vulnerabilities.

Recommendations. Using macros, or macro-based libraries like Serde (which has a `#[serde(skip)]` helper to avoid serialization redundant fields) reduce human errors by reducing the amount of manual (de)serialization code that has to be written or reviewed.

In addition, avoiding serialization of redundant fields forces the deserialization to always recompute these fields, which in turn reduces the risk of deserialization bugs where someone forgets to check that these fields contain the correct values.