AlephBFT

Security Assessment

June 21, 2021

Prepared For:
Adam Gągol | *Cardinal Cryptography & Aleph Zero Foundation*
adam.gagol@cardinals.cc

Prepared By:
Artur Cygan | *Trail of Bits*
artur.cygan@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Changelog:

June 21, 2021:	Initial report draft
July 27, 2021:	Added Appendix D: Fix Log

Executive Summary

From June 7 to June 18, 2021, the Aleph Zero Foundation engaged Trail of Bits to review the security of AlephBFT, a Rust implementation of the Aleph Zero consensus protocol. Trail of Bits conducted this assessment over four person-weeks, with two engineers working from commit hash b73a6caf from the AlephBFT repository.

During the first week of the engagement, we focused on gaining an understanding of the codebase and reading the documentation and the Aleph Zero paper. We looked for common Rust security flaws and started the review of the Terminal, Reliable Multicast, and Alerts components. During the second week, we performed a manual review of the other components, focusing on the Member and Consensus. We also checked the data serialization routines and the interfaces exposed by the AlephBFT library.

Our review resulted in nine findings ranging from low to informational severity. The low-severity finding involves the handling of errors in the asynchronous code, which could lead to an inconsistent node state in the event of a panic. The informational-severity findings pertain to minor issues that currently have no measurable security impact on the system; however, addressing those findings would further harden the protocol.

AlephBFT ships with comprehensive, up-to-date documentation describing the implementation and how it relates to the paper. Additionally, the code contains an extensive test suite, covering scenarios such as unreliable networks and Byzantine nodes and including random testing akin to property-based testing. Nevertheless, we found a deficiency in negative testing, resulting in a lack of coverage of certain code paths (TOB-ALEPH-004); the AlephBFT code also lacks fuzzing harnesses for incoming data, which would increase all stakeholders' confidence in the protocol's reliability.

As AlephBFT abstracts away many critical components, such as signing, key management, and networking processes, it is not possible to reason about the security of the end system as a whole by assessing only AlephBFT. We therefore recommend performing additional security assessments of the systems into which AlephBFT is integrated.

Project Dashboard

Application Summary

Name	AlephBFT
Version	b73a6caf
Type	Rust
Platform	Native

Engagement Summary

Dates	June 7 – June 18, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total Low-Severity Issues	1	■
Total Informational-Severity Issues	8	■■■■■■■■
Total	9	

Category Breakdown

Data Validation	2	■■
Denial of Service	1	■
Error Reporting	3	■■■
Patching	1	■
Undefined Behavior	2	■■
Total	9	

Engagement Goals

The engagement was scoped to provide a security assessment of AlephBFT, a Rust implementation of the Aleph Zero consensus protocol.

Specifically, we sought to answer the following questions:

- Is it possible to crash a node by using certain crafted input?
- Is it possible to spam the protocol to cause a denial of service?
- Would an attack prevent honest nodes from producing output?
- Are there any situations in which nodes could reach different conclusions on a matter?
- If the number of Byzantine nodes is below the assumed threshold, will the consensus protocol operate as expected?
- Does the implementation match that outlined in the documentation and the paper?
- Does the Aleph Zero protocol adhere to its proposed Byzantine fault tolerance?

Coverage

Documentation. We spent the first two days of the audit reading the documentation included in the GitHub repository and the Aleph Zero protocol paper to familiarize ourselves with the protocol. We also assessed the documentation's quality.

AlephBFT. We performed an in-depth manual review of the Rust code, looking for discrepancies between the documentation and the paper. We also analyzed the quality of the test suite, worked to identify any fuzz targets, and looked for common Rust mistakes (using static analysis tools such as `cargo-audit` and `rust-clippy`). Lastly, using `cargo-fuzz`, we developed a fuzzing harness to validate the serialization of data structures.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Add comments to the AlephBFT code referencing specific sections or paragraphs of the paper and noting any deviations from the specification.** TOB-ALEPH-001

❑ **Define the `Round` type as `u16` and use that type in all code that deals with rounds.** Upcast the `Round` type to a `usize` value when accessing vector elements. Alternatively, change the casting process to include a check for overflows and to terminate the program if an overflow occurs. This will prevent the program from continuing to execute with corrupted data. TOB-ALEPH-002

❑ **Either provide a label for the outer loop and use it to break out of that loop or use a return statement.** That way, the code will work as intended. TOB-ALEPH-003

❑ **Ensure that the `self.n_units_per_round` state is decremented when a forker's units are removed from the store.** TOB-ALEPH-004

❑ **Enable the hook mechanism to return a `Result` and to take appropriate action in the `update_on_dag_add` function.** Update the registered hooks such that they return `Err` when an `unbounded_send` error occurs. TOB-ALEPH-005

❑ **Change the `BoolNodeMap` `Decode` implementation to use only the canonical encoding.** TOB-ALEPH-006

❑ **Modify the `SpawnHandle` trait to return a task handle and ensure that parent tasks wait for all child tasks they have spawned, in addition to the exit signal.** If a child task exits with an unrecoverable error (e.g., one that would make restarting the task pointless), the parent task should terminate the other child tasks and exit. Lastly, ensure that shutdowns are propagated upward. TOB-ALEPH-007

❑ **Either mark `send` and `broadcast` as async or document the `Network` trait, explaining that the `send` and `broadcast` implementations must not block.** TOB-ALEPH-008

❑ **Choose one of the two methods of handling closed channel errors and use it wherever possible; if it would make more sense to use the other in certain cases, document that decision.** TOB-ALEPH-009

Long Term

❑ **Extensively document complex code.** TOB-ALEPH-001

❑ **Use types that adhere as closely as possible to the data semantics.** TOB-ALEPH-002

❑ **Make sure that all branches of the code are covered by the test suite.** TOB-ALEPH-003, TOB-ALEPH-004, TOB-ALEPH-005

❑ **Add a fuzzing harness or property test to ensure that the result of composing the `decode` and `encode` functions is an identity function.** TOB-ALEPH-006

❑ **Add tests to make sure that the system shuts down correctly when one of its tasks panics.** TOB-ALEPH-007

❑ **Be mindful of the fact that blocking functions in async code may introduce subtle performance issues that lead to downtime.** TOB-ALEPH-008

❑ **Ensure that the codebase handles errors consistently.** This will make the code easier to develop and understand. TOB-ALEPH-009

Findings Summary

#	Title	Type	Severity
1	Code documentation does not reference the paper	Patching	Informational
2	Use of different types to represent rounds	Data Validation	Informational
3	Use of incorrect loop break to handle add_to_store and handle_events failures	Undefined Behavior	Informational
4	Incorrect state rollback upon removal of forker's units	Undefined Behavior	Informational
5	Lack of error handling in Terminal's post-insert hooks	Error Reporting	Informational
6	Different byte representations decode to the same data	Data Validation	Informational
7	Errors in async code leave the program in an inconsistent state	Error Reporting	Low
8	Blocking I/O in Network trait implementations will block async runtime threads	Denial of Service	Informational
9	Inconsistent handling of closed channel errors	Error Reporting	Informational

1. Code documentation does not reference the paper

Severity: Informational	Difficulty: N/A
Type: Patching	Finding ID: TOB-ALEPH-001
Target: `AlephBFT`	

Description
The AlephBFT Rust code implements the consensus protocol outlined in the Aleph Zero paper; however, it does not reference specific sections of the paper, which makes it more difficult to audit the code. For example, specific references to the “Practical Considerations” section, which details many modifications and improvements made to the protocol, would be beneficial.

Recommendations
Short term, add comments to the AlephBFT code referencing specific sections or paragraphs of the paper and noting any deviations from the specification.

Long term, extensively document complex code.

2. Use of different types to represent rounds

Severity: Informational
Type: Data Validation
Target: `AlephBFT`

Difficulty: High
Finding ID: TOB-ALEPH-002

Description

The code uses different types to represent rounds in the protocol (figures 2.1 and 2.2). It primarily uses the `Round` type, which is defined as a `usize` type. However, parts of the protocol code (like that in figure 2.3) do not use this type, which is non-ideal. When a `UnitCoord` is created, a round is passed in as `Round` (`usize`) and downcast to a `u16` value (figure 2.2). If the round number is larger than `u16`, the program will continue running with an incorrect round number (specifically, the remainder of the downcasting operation). However, because the current maximum number of rounds is set rather low, none of the round numbers will be larger than `u16`.

```rust
/// An asynchronous round of the protocol.
pub type Round = usize;
```

Figure 2.1: [src/lib.rs#L76-L77]()

```rust
#[derive(Debug, Clone, Copy, Eq, PartialEq, Encode, Decode, StdHash)]
pub(crate) struct UnitCoord {
    pub(crate) round: u16,
    creator: NodeIndex,
}

impl UnitCoord {
    pub fn new(round: Round, creator: NodeIndex) -> Self {
        Self {
            creator,
            round: round as u16,
        }
    }

    pub fn creator(&self) -> NodeIndex {
        self.creator
    }

    pub fn round(&self) -> Round {
        self.round as Round
    }
}
```

Figure 2.2: [src/units.rs#L10-L31]()

```rust
pub(crate) struct UnitStore<'a, H: Hasher, D: Data, KB: KeyBox> {
    by_coord: HashMap<UnitCoord, SignedUnit<'a, H, D, KB>>,
    by_hash: HashMap<H::Hash, SignedUnit<'a, H, D, KB>>,
    parents: HashMap<H::Hash, Vec<H::Hash>>,
    //this is the smallest r, such that round r-1 is saturated, i.e., it has at least
threshold (~(2/3)N) units
    round_in_progress: usize,
    threshold: NodeCount,
    //the number of unique nodes that we hold units for a given round
    n_units_per_round: Vec<NodeCount>,
    is_forker: NodeMap<bool>,
    legit_buffer: Vec<SignedUnit<'a, H, D, KB>>,
    max_round: usize,
}
```

Figure 2.3: [src/units/store.rs#L3-L15]

Recommendations

Short term, define the `Round` type as `u16` and use that type in all code that deals with rounds. Upcast the `Round` type to a `usize` value when accessing vector elements. Alternatively, change the casting process to include a check for overflows and to terminate the program if an overflow occurs. This will prevent the program from continuing to execute with corrupted data.

Long term, use types that adhere as closely as possible to the data semantics.

3. Use of incorrect loop break to handle add_to_store and handle_events failures

Severity: Informational
Type: Undefined Behavior
Target: `AlephBFT Terminal`

Difficulty: High
Finding ID: TOB-ALEPH-003

Description

The Terminal code that executes event handlers (figure 3.1) uses a break to stop the processing of events when an error occurs. However, because the break stops the `for` loop instead of the outer loop, the events will continue to be processed. This could lead to an undesirable node state.

Currently, the error can be caused only by a closed channel, which is considered a programming mistake that should not ever happen. However, further developments in the AlephBFT code could introduce additional error conditions more likely to trigger this issue.

```
pub(crate) async fn run(&mut self, mut exit: oneshot::Receiver<()>) {
    loop {
        futures::select! {
            n = self.ntfct_rx.next() => {
                match n {
                    Some(NotificationIn::NewUnits(units)) => {
                        for u in units {
                            if self.add_to_store(u).is_err() ||
self.handle_events().is_err() {
                                break
                            };
                        }
                    },
                    // (...)
                }
            }
            // (...)
        }
    }
}
```

Figure 3.1: src/terminal.rs#L352-L379

Recommendations

Short term, either provide a label for the outer loop and use it to break out of that loop or use a return statement. That way, the code will work as intended.

Long term, make sure that all branches of the code are covered by the test suite.

References

- Rust By Example: Nesting and labels

4. Incorrect state rollback upon removal of forker’s units

Severity: Informational	Difficulty: High
Type: Undefined Behavior	Finding ID: TOB-ALEPH-004
Target: `AlephBFT Unit Store`	

Description
In the `mark_forker` function, any of the forker’s units with round numbers larger than that of the round in progress are removed from the store (figure 4.1). This operation should revert the store’s state, making it as though the store never contained those units. However, this does not actually occur; instead, the number of units in the `self.n_units_per_round` state is incremented as units are added (figure 4.2), but it is not decremented by `mark_forker`.

The error does not currently impact the correctness of AlephBFT.

```
for round in self.round_in_progress + 1..=self.max_round {
    let coord = UnitCoord::new(round, forker);
    if let Some(su) = self.unit_by_coord(coord).cloned() {
        // (...)
        self.by_coord.remove(&coord);
        let hash = su.as_signable().hash();
        self.by_hash.remove(&hash);
        self.parents.remove(&hash);
        // Now we are in a state as if the unit never arrived.
    }
}
```

Figure 4.1: src/units/store.rs#L115-L119

```
if self.by_coord.insert(coord, su.clone()).is_none() {
    // This means that this unit is not a fork (even though the creator might be a forker)
    self.n_units_per_round[round] += NodeCount(1);
}
```

Figure 4.2: src/units/store.rs#L146-L149

Recommendations
Short term, ensure that the `self.n_units_per_round` state is decremented when a forker’s units are removed from the store.

Long term, make sure that all branches of the code are covered by the test suite.

5. Lack of error handling in Terminal's post-insert hooks

Severity: Informational
Type: Error Reporting
Target: `AlephBFT`

Difficulty: High
Finding ID: TOB-ALEPH-005

Description

The Terminal allows users to register post-insert hooks that will be executed when a unit is added to the DAG (figure 5.1). The handler type does not include a means of dealing with a failure, and the program will continue regardless of the outcome of running a handler. There are two hooks registered in the `Consensus::run` function. Both of them can fail when sending data to a channel; however, any errors will be effectively silenced (figure 5.2).

```
fn update_on_dag_add(&mut self, u_hash: &H::Hash) -> Result<(), ()> {
    // (...)
    self.post_insert.iter().for_each(|f| f(u.clone()));
    // (...)
}
```

Figure 5.1: src/terminal.rs#L233-L255

```
// send a new parent candidate to the creator
terminal.register_post_insert_hook(Box::new(move |u| {
    if let Err(e) = parents_tx.unbounded_send(u.into()) {
        debug!(target: "AlephBFT", "channel to creator is closed {:?}", e);
    }
}));
// try to extend the partial order after adding a unit to the dag
terminal.register_post_insert_hook(Box::new(move |u| {
    if let Err(e) = electors_tx.unbounded_send(u.into()) {
        debug!(target: "AlephBFT", "channel to extender is closed {:?}", e);
    }
}));
```

Figure 5.2: src/consensus.rs#L44-L55

Recommendations

Short term, enable the hook mechanism to return a `Result` and to take appropriate action in the `update_on_dag_add` function. Update the registered hooks such that they return `Err` when an `unbounded_send` error occurs.

Long term, make sure that all branches of the code are covered by the test suite.

6. Different byte representations decode to the same data

Severity: Informational
Type: Data Validation
Target: `AlephBFT`

Difficulty: High
Finding ID: TOB-ALEPH-006

Description

Data structures in AlephBFT use Parity SCALE Codec for serialization. The `BoolNodeMap` structure has custom `Encode` and `Decode` trait implementations (figure 6.1). The implementation of the `Decode` trait allows for the use of different byte representations, since the trait does not check whether the decoded `capacity` reflects the number of bytes. The `BitVec` is simply truncated to the decoded value, which has no effect when the `capacity` exceeds the `BitVec` length. `BoolNodeMap` is deeply embedded in the `Unit` structure.

The use of different byte representations for the same data could confuse honest nodes; for example, if a signature were checked against the byte representation and later re-encoded into different bytes, the signature would no longer match the byte representation. AlephBFT is not affected by this issue, since bytes are decoded and then encoded before signatures are checked against the canonical representation.

We found this issue by fuzzing the decoding cycle, as detailed in Appendix C.

```
impl Encode for BoolNodeMap {
    fn encode_to<T: Output + ?Sized>(&self, dest: &mut T) {
        (self.0.len() as u32).encode_to(dest);
        self.0.to_bytes().encode_to(dest);
    }
}

impl Decode for BoolNodeMap {
    fn decode<I: Input>(input: &mut I) -> Result<Self, Error> {
        let capacity = u32::decode(input)? as usize;
        let bytes = Vec::decode(input)?;
        let mut bv = bit_vec::BitVec::from_bytes(&bytes);
        bv.truncate(capacity);
        Ok(BoolNodeMap(bv))
    }
}
```

Figure 6.1: src/nodes.rs#L126-L141

Recommendations

Short term, change the `BoolNodeMap` `Decode` implementation to use only the canonical encoding.

Long term, add a fuzzing harness or property test to ensure that the result of composing the `decode` and `encode` functions is an identity function.

7. Errors in async code leave the program in an inconsistent state

Severity: Low
Type: Error Reporting
Target: AlephBFT

Difficulty: High
Finding ID: TOB-ALEPH-007

Description

Most of the AlephBFT code is executed asynchronously through a hierarchy of tasks. While AlephBFT does not enforce the use of a specific async runtime, the most popular choice is the Tokio runtime, also used in AlephBFT tests. When code running in a Tokio task calls `panic`, it will default to stopping the task, and the resulting `JoinHandle` will return an `Err`. By contrast, in non-async Rust code, a call to `panic` will terminate the whole program and lead to an unexpected program state.

The panicking functions `unwrap` and `expect` are used throughout the code with the assumption that they will terminate the program. However, a task termination does not initiate a system shutdown, as parent tasks do not monitor or wait for the child tasks that they spawn. The Consensus task, for example, spawns Extender, Creator, and Terminal tasks, but the `spawn` function of the `SpawnHandle` trait does not return a handle. The Consensus task assumes that all child tasks it has spawned will execute correctly and waits only for a shutdown message (figure 7.1). The same pattern is present in the Member code.

```
pub(crate) async fn run<H: Hasher + 'static>(/* (...) */) {
    debug!(target: "AlephBFT", "{:?} Starting all services...", conf.node_ix);

    let n_members = conf.n_members;

    let (electors_tx, electors_rx) = mpsc::unbounded();
    let mut extender = Extender::<H>::new(conf.node_ix, n_members, electors_rx,
ordered_batch_tx);
    let (extender_exit, exit_rx) = oneshot::channel();
    spawn_handle.spawn("consensus/extender", async move {
        extender.extend(exit_rx).await
    });

    let (parents_tx, parents_rx) = mpsc::unbounded();
    let new_units_tx = outgoing_notifications.clone();
    let mut creator = Creator::new(conf.clone(), parents_rx, new_units_tx);

    let (creator_exit, exit_rx) = oneshot::channel();
    spawn_handle.spawn(
        "consensus/creator",
        async move { creator.create(exit_rx).await },
    );

    let mut terminal = Terminal::new(conf.node_ix, incoming_notifications,
outgoing_notifications);

    // (...)

    let (terminal_exit, exit_rx) = oneshot::channel();
    spawn_handle.spawn(
        "consensus/terminal",
        async move { terminal.run(exit_rx).await },
    );
    debug!(target: "AlephBFT", "{:?} All services started.", conf.node_ix);

    let _ = exit.await;
    // we stop no matter if received Ok or Err
    let _ = terminal_exit.send(());
    let _ = creator_exit.send(());
    let _ = extender_exit.send(());

    debug!(target: "AlephBFT", "{:?} All services stopped.", conf.node_ix);
}
```

Figure 7.1: src/consensus.rs#L13-L71

The `move_units_to_consensus` function also contains an unsupervised task (figure 7.2). Although the task is not short-lived, the same concerns apply.

```
fn move_units_to_consensus(&mut self) {
    for su in self.store.yield_buffer_units() {
        let full_unit = su.as_signable();
        let unit = full_unit.unit();
        if let Some(avail_fut) = self.data_io.check_availability(full_unit.data()) {
            let tx_consensus = self.tx_consensus.clone();
            self.spawn_handle
                .spawn("member/check_availability", async move {
                    // (...)
                });
        } else {
            self.send_consensus_notification(NotificationIn::NewUnits(vec![unit]))
        }
    }
}
```

Figure 7.2: src/member.rs#L425-L445

It is possible to modify the panic behavior by configuring the program to abort upon a panic; however, this might not be beneficial to the end binary using AlephBFT.

Exploit Scenario

A panic occurs in the async code; the Tokio task it is running on is terminated, and the other tasks keep running. This violates the assumption that the entire program will be terminated and causes the program to enter an unexpected state.

Recommendations

Short term, modify the `SpawnHandle` trait to return a task handle and ensure that parent tasks wait for all child tasks they have spawned, in addition to the exit signal. If a child task exits with an unrecoverable error (e.g., one that would make restarting the task pointless), the parent task should terminate the other child tasks and exit. Lastly, ensure that shutdowns are propagated upward.

Long term, add tests to make sure that the system shuts down correctly when one of its tasks panics.

References

- Module `tokio::task`: Working with tasks
- Aborting on panic

8. Blocking I/O in Network trait implementations will block async runtime threads

Severity: Informational
Type: Denial of Service
Target: AlephBFT

Difficulty: High
Finding ID: TOB-ALEPH-008

Description

The Network trait specifies send and broadcast functions that are not marked as async (figure 8.1). If the trait is implemented with blocking versions of the send and broadcast functions, calls to those functions will block async runtime threads. The AlephBFT user is responsible for the implementation of the trait and may not be aware of this issue. A sample implementation of the Network trait in the test code does not block, since it uses channels to store data for processing.

```
#[async_trait::async_trait]
pub trait Network<H: Hasher, D: Data, S: Signature, MS: PartialMultisignature>: Send {
    type Error: Debug;
    /// Send a message to a single node.
    fn send(&self, data: NetworkData<H, D, S, MS>, node: NodeIndex) -> Result<(),
Self::Error>;
    /// Send a message to all nodes.
    fn broadcast(&self, data: NetworkData<H, D, S, MS>) -> Result<(), Self::Error>;
    /// Receive a message from the network.
    async fn next_event(&mut self) -> Option<NetworkData<H, D, S, MS>>;
}
```

Figure 8.1: src/network.rs#L31-L40

Recommendations

Short term, either mark send and broadcast as async or document the Network trait, explaining that the send and broadcast implementations must not block.

Long term, be mindful of the fact that blocking functions in async code may introduce subtle performance issues that lead to downtime.

9. Inconsistent handling of closed channel errors

Severity: Informational	Difficulty: High
Type: Error Reporting	Finding ID: TOB-ALEPH-009
Target: `AlephBFT`	

Description

The code handles closed channel errors in two ways: by using `expect` (figure 9.1) and by returning `Err` (figure 9.2).

```
        self.multisigned_hashes_tx
            .unbounded_send(multisigned.clone())
            .expect("Channel should be open");
```

Figure 9.1: src/rmc.rs#L230-L232

```
        self.finalizer_tx
            .unbounded_send(batch)
            .map_err(|e| {
                debug!(target: "AlephBFT-extender", "{:?} channel for batches is closed
{:?}, closing", self.node_id, e);
            })?;
```

Figure 9.2: src/extender.rs#L151-L155

Recommendations

Short term, choose one of the two methods of handling closed channel errors and use it wherever possible; if it would make more sense to use the other in certain cases, document that decision.

Long term, ensure that the codebase handles errors consistently. This will make the code easier to develop and understand.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.
Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.

High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is commonly exploited; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of a complex system.
High	An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Quality Recommendations

This appendix contains findings that do not have immediate or obvious security implications.

- The `is_new_fork` function could take `FullUnit` as an argument instead of `SignedUnit`.
- The loop that recomputes votes in the Extender task could use a labeled outer loop and could break as soon as a decision is reached instead of repeating the breaking code.
- Using a `match(decision) { …}` block in the code that handles voting decisions would increase the code's clarity.
- There is a typo in the log messages in the network hub: the "n" is missing from `AlephBFT-network-hub`.
- There are typos in the assert messages in the `decoding_network_data_units_new_unit` test.
- The empty comment before the `initialize_round` function should be removed.

C. Fuzzing the Custom Serialization Roundtrip

Using the rust-fuzz/cargo-fuzz crate, Trail of Bits developed a fuzzing harness to test the AlephBFT type serialization functions. This led us to discover that there are multiple byte representations in the code that decode to the same `BoolNodeMap` object, which could be problematic in certain cases (TOB-ALEPH-006).

Figure C.1 shows the fuzzing harness that we implemented. There are certain changes that will need to be made to run this fuzzing harness (e.g., making certain types public and using the file structure required by `cargo-fuzz`). These changes are included in the patch provided with this report, which was developed on commit `b73a6ca` and can be added to the code through the `git apply <patchfile>` command.

To launch the harness, use the `cargo fuzz run fuzz-codec` command. When it finds a decoding roundtrip issue, it will display the raw bytes used for decoding, the encoded objects' representation, and the leftover data that was not decoded (the variable `d`). This information may be displayed as follows:

```
Type: aleph_bft::nodes::BoolNodeMap
d (leftover/undecoded data) = []
- Encoded data:
raw1       = [0, 36, 0, 0, 0]
raw2       = [0, 0, 0, 0, 0]
- Decoded objects:
obj1       = 'BoolNodeMap()'
obj2       = 'BoolNodeMap()'
thread '<unnamed>' panicked at 'raw1 != raw2', fuzz_targets/fuzz-codec.rs:42:25
```

The patch also contains a `fuzz/fuzz_targets/main.rs` file that can be used to debug given input (through an IDE, for example). It includes a few inputs that cause multiple byte representations to be decoded to the same object.

```
#![no_main]
use libfuzzer_sys::fuzz_target;

use std::{
    collections::hash_map::DefaultHasher,
    hash::Hasher,
    fmt::Debug,
    cmp::PartialEq
};
use aleph_bft::{
    units::{ControlHash, UnitCoord},
    nodes::BoolNodeMap
};
// Note: set in Cargo.toml to be the same parity-scale-codec as used by AlephBFT
use codec::{Decode, Encode};
```

```
fn fuzz_codec_roundtrip<T: Decode + Encode + Debug + PartialEq>(mut data: &[u8]) {
    let raw1 = data.clone();
    let maybe_object = <T>::decode(&mut data);

    // Skip if the input failed to decude
    if let Ok(object) = maybe_object {
        let mut data2: &[u8] = &object.encode();
        let raw2 = data2.clone();

        let expected_object = <T>::decode(&mut data2);

        match expected_object {
            Ok(object2) => {
                if object == object2 {
                    // Note: we ignore the leftover/not decoded bytes
                    let bytes_decoded = raw1.len() - data.len();
                    if &raw1[..bytes_decoded] != raw2 {
                        println!("Type: {}", std::any::type_name::<T>());
                        println!("d (leftover/undecoded data) = {:?}", data);
                        println!("- Encoded data:");
                        println!("raw1       = {:?}", raw1);
                        println!("raw2       = {:?}", raw2);
                        println!("- Decoded objects:");
                        println!("obj1       = '{:?}'", object);
                        println!("obj2       = '{:?}'", object2);
                        panic!("raw1 != raw2");
                    }
                    return
                }
                panic!("obj != obj2; obj={:?}, obj2={:?}", object, object2);
            }
            Err(e) => {
                panic!("Cannot .decode().encode().decode() - is that a bug? err: {}", e);
            }
        }
    };
}

// Hasher64 -- not cryptographically secure, mocked for demonstration purposes
// (taken from AlephBFT examples)
#[derive(PartialEq, Eq, Clone, Debug)]
struct Hasher64;

impl aleph_bft::Hasher for Hasher64 {
    type Hash = [u8; 8];
    fn hash(x: &[u8]) -> Self::Hash {
        let mut hasher = DefaultHasher::new();
        hasher.write(x);
        hasher.finish().to_ne_bytes()
    }
}

fuzz_target!(|data: &[u8]| {
    //fuzz_codec_roundtrip::<ControlHash::<Hasher64>>(data);
    //fuzz_codec_roundtrip::<UnitCoord>(data); // no trigger?
    fuzz_codec_roundtrip::<BoolNodeMap>(data);
});
```

Figure C.1: A fuzzing harness used to test custom AlephBFT type encoders.

D. Fix Log

The Aleph Zero team has addressed the following issues in their codebase as a result of our assessment, and each of the fixes was verified by Trail of Bits.

After completion of the initial assessment, Aleph Zero team addressed the discovered issues in a series of already merged pull requests (,

#	Title	Severity	Status
1	Code documentation does not reference the paper	Informational	Fixed
2	Use of different types to represent rounds	Informational	Fixed
3	Use of incorrect loop break to handle add_to_store and handle_events failures	Informational	Fixed
4	Incorrect state rollback upon removal of forker's units	Informational	Fixed
5	Lack of error handling in Terminal's post-insert hooks	Informational	Fixed
6	Different byte representations decode to the same data	Informational	Fixed
7	Errors in async code leave the program in an inconsistent state	Low	Fixed
8	Blocking I/O in Network trait implementations will block async runtime threads	Informational	Fixed
9	Inconsistent handling of closed channel errors	Informational	Fixed

For additional information for each fix, please refer to the detailed fix log below.

Detailed fix log

TOB-ALEPH-001: Code documentation does not reference the paper
Fixed in PR #110. The code is now commented with references to the Aleph Zero paper and the implementation documentation.

TOB-ALEPH-002: Use of different types to represent rounds

Fixed in PR #100. The code was changed to use the Round type everywhere, casting values to the usize type when accessing vector elements.

TOB-ALEPH-003: Use of incorrect loop break to handle add_to_store and handle_events failures
Fixed in PR #101. The code was changed to use a return statement instead of break.

TOB-ALEPH-004: Incorrect state rollback upon removal of forker's units
Fixed in PR #102. The missing round decrementation was added along with tests for the state rollback.

TOB-ALEPH-005: Lack of error handling in Terminal's post-insert hooks
Fixed in PR #106. The hooks were changed to use panic instead of simply logging that an error occurred.

TOB-ALEPH-006: Different byte representations decode to the same data
Fixed in PR #106. We ran the fuzzing harness from the Appendix C on the updated code and didn't find any different examples which decode to the same data.

TOB-ALEPH-007: Errors in async code leave the program in an inconsistent state
Fixed in PR #109. The spawn_essential function was added to the SpawnHandle trait which returns a task handle. The handle is used to ensure that the error handling is propagated to parent tasks.

TOB-ALEPH-008: Blocking I/O in Network trait implementations will block async runtime threads
Fixed in PR #103. Documentation was added to the Network trait warning implementers about blocking I/O in send and broadcast function implementations.

TOB-ALEPH-009: Inconsistent handling of closed channel errors
Fixed in PR #108. The error handling of closed channels is now consistent across the codebase and uses panic.