# **Improve Floresta’s Testing**

[Floresta](https://github.com/Davidson-Souza/Floresta)

Abhinav Agarwalla

Indian Institute of Technology BHU, Varanasi, India

## 

## **Name and Contact Information**

**Name**: Abhinav Agarwalla

**Email**: [abhinavagarwalla6@gmail.com](mailto:abhinavagarwalla6@gmail.com)

**Discord** **username**: lla\_dane

**Linkedin**: [Abhinav Agarwalla](https://www.linkedin.com/in/abhinav-agarwalla-a80425258/)

**Github**: [lla\_dane](https://github.com/lla-dane)

**Location**: India

**Timezone**: Kolkata, India, UTC+5:30

**Typical** **working** **hours**: 9 – 10 hours between 10 AM to 11 PM

## **About Me**

**Education**

* **University:** Indian Institute of Technology, Varanasi, India
* **Major:** Mathematics and Computing
* **Degree Leve**l: 2nd year undergraduate
* **Graduation Year**: 2027

### **Why is this project important to me?**

This project will primarily deal with the development and implementation of testing strategy that includes unit testing, integration testing, functional testing, performance testing, etc. Given the project’s focus I’ll deepen my understanding of Bitcoin and blockchain technology. The insights gained from understanding how blockchain works – how new blocks are mined, how consensus is achieved, and most interestingly how the peer-to-peer communication network without needing a central server is implemented, is genuinely fascinating.

### **Why me?**

I very much enjoy learning and using new technologies in core computer science. After coming to college, I explored different fields like **game development, 3D modelling, web development** etc. I started learning about blockchain through the **Summer of Bitcoin** bootcamp, and it fascinated me the most about how complex and amazing the core mechanics of **Bitcoin** are, and I plan on learning and making an impact to my full potential in this project and improving my knowledge about **Bitcoin** and overall, Web 3.0. Having previously worked on rust projects, makes it the right choice of project for me.

## **Competency Test**

Build Floresta from source and run existing functional and unit tests in your machine

To run the tests successfully, I ran the following commands in sequence:

* ` **cargo build --release --bin florestad** `
* ` **cargo install --path ./florestad** `
* Replaced the config.toml.sample with config.toml with the following contents which was present the tutorial docs in **florestad**:

[wallet]

xpubs = [

"vpub5ZkWR4krsAzwFsmW8Yp3eHuANVLr7ZSWii6KmRnLRiN6ZXLbqs1f217jJM37oteQoyng82yw44XQU8PYJJBGgVzvJ96dQZEyZZcDiDmoJXw",

"vpub5V5XF4ipcQ9tLp7NCFswnwZ23tm5Key81E9CCfqFXaGjzTpQ8jjiirf2hG7aXtqXbRDFxMvEhdGdeFcqQ3jUGUkq4mqo2VoGCDWCZvPQvUy",

]

addresses = [

"tb1qjfplwf7a2dpjj04cx96rysqeastvycc0j50cch"

]

descriptors = [

"wsh(sortedmulti(1,[54ff5a12/48h/1h/0h/2h]tpubDDw6pwZA3hYxcSN32q7a5ynsKmWr4BbkBNHydHPKkM4BZwUfiK7tQ26h7USm8kA1E2FvCy7f7Er7QXKF8RNptATywydARtzgrxuPDwyYv4x/<0;1>/\*,[bcf969c0/48h/1h/0h/2h]tpubDEFdgZdCPgQBTNtGj4h6AehK79Jm4LH54JrYBJjAtHMLEAth7LuY87awx9ZMiCURFzFWhxToRJK6xp39aqeJWrG5nuW3eBnXeMJcvDeDxfp/<0;1>/\*))#fuw35j0q"

]

* Ran ` **florestad -c config.toml --network signet run**`to run the node in signet.
* In the `**./crates/floresta-cli`** ran **`cargo build --release`**
* Then ran **`cargo build --release`**in the source directory.
* Then ran **`cargo test`**and all the tests ran successfully
* Checkout the checks running successfully in this [**clip**](https://drive.google.com/file/d/1H4RP2CDgTChWAlsPkYKZXBLBUT8hTKIg/view?usp=sharing)**.**

## **Skills aligned with the project**

I have been doing a lot of rust programming in recent times:

* **SOB Assignment:** We were assigned with a problem statement where we were given a mempool of real transactions, and we had to mine a block while correctly validating all transactions according to the **Bitcoin Consensus Rules** without using any bitcoin related libraries**.** I did this assignment on **rust** as it gave me great low-level access and **memory-optimizations** in my code, which is difficult in other programming languages. Our score was calculated based on how efficiently we accommodate transactions with high gas-fees in the block. I got a score of **101/100** as I increased the maximum expected gas-fees in the block. It gave me a deep understanding of the logic behind how transactions are validated in **Bitcoin.**
* [**Rustchain**](https://github.com/lla-dane/Rust-Chain)**:** While going through the SOB bootcamp where we have to go through 2 chapters of **GROKKING BITCOIN,** every week, meanwhile I followed a [**tutorial**](https://www.youtube.com/playlist?list=PLc0PxFU2AtMQJ0ocblyewzWG60k6vLzLL) in which I built a simplified implementation of bitcoin based blockchain system. It gave me great hands-on experience with Bitcoin based data-structures and p2p implementations. I am working on implementing some of my own features in it.
* I also played around with some simple [**rust programs**](https://github.com/lla-dane/Rusties)while I was following the [**Rust-lang book**](https://doc.rust-lang.org/book/) for basics.

## **Synopsis**

The aim of this project is to enhance the testing infrastructure of Floresta, ensuring the delivery of high-quality, secure, and privacy-focused full-node software. The goal of the project is to build a comprehensive testing suite that minimizes bugs and vulnerabilities, improving the overall assurance and reliability of the software.

The testing infrastructure can be divided into the following parts:

1) Unit Testing

2) Integration Testing

3) Performance Testing

4) Security Testing

**Mentors**

* Davidson Souza

## **Plan of Action**

### 1. **Unit testing:**

For unit testing it is necessary to identify with part of the code is not included in the tests.

For expanding the unit testing infrastructure of **Floresta** I created an analysis report mechanism using ***‘cargo-tarpaulin’***, it is a popular code coverage tool specifically designed for Rust projects. It provides an easy way to assess the effectiveness of your tests by measuring the code coverage, indicating which parts of your codebase are executed during tests.

Check out the present testing analysis report of **Floresta** [here](https://lla-dane.github.io/floresta-tests.github.io/) with **22.04%** of the codebase covered. The ***floresta-cli*** tests are not included in the report.

This crate will efficiently help me to cover the codebase with unit tests.

### **2. Integration Testing:**

An efficient integration testing infrastructure ensures that the components (**Bitcoin node, Utreexo and Electrum server**) work together seamlessly.

Below are some key integration test ideas that I came up with:

* **Receiving data from other peers**:

For **simulating interactions** with other **Bitcoin nodes** and test how our node responds to incoming blocks, transactions and network messages,we can mock the behavior of peers sending incorrect data, attempting double spends, or acting maliciously to test the resilience and error handling of the system.

**How can we do this?**

Libraries such as **`tokio`** for asynchronous runtime and **`mockall`** and **`async\_trait`** for mocking traits will be essential.

* [**Tokio**](https://docs.rs/tokio/latest/tokio/)**:** Tokio is an asynchronous runtime for Rust. It is used to handle async operations such as network communications, timeouts, and other I/O operations.
* [**Mockall**](https://docs.rs/mockall/latest/mockall/)**:** A powerful and flexible library for creating mock objects in Rust tests, it's particularly useful for unit and integration testing, to simulate the behavior of complex components without relying on their actual implementations.
* [**Async\_trait**](https://docs.rs/async-trait/latest/async_trait/)**:** It enables defining **`async`** methods within traits, something not supported natively by Rust. In the below example, since **`AsyncRead`** and **`AsyncWrite`** require the implementation of **`poll\_read`** and **`poll\_write`** methods which are inherently asynchronous, **`async\_trait`** makes it possible to define these methods cleanly in a trait.

**Define Mocks:**

We will need to create mock objects that can simulate the behavior of real network components, such as the Transport layer and perhaps the Bitcoin network’s messaging system.

Here’s an example of how we might mock a **`Transport`** trait to simulate network interactions:

use mockall::{automock, predicate::\*};

use async\_trait::async\_trait;

use futures::io::{AsyncRead, AsyncWrite};

#[automock]

pub trait Transport: AsyncRead + AsyncWrite + Unpin + Clone + Sync + Send + 'static {

// Add methods that need to be mocked

fn shutdown(&mut self) -> std::result::Result<(), std::io::Error>;

}

#[async\_trait]

impl AsyncRead for MockTransport {

// This function simulates the async read operation.

async fn poll\_read(

self: Pin<&mut Self>,

cx: &mut Context<'\_>,

buf: &mut [u8]

) -> Poll<std::result::Result<usize, std::io::Error>> {

AsyncRead::poll\_read(self, cx, buf)

}

}

#[async\_trait]

impl AsyncWrite for MockTransport {

// This function simulates the async write operation.

async fn poll\_write(

self: std::pin::Pin<&mut Self>,

cx: &mut std::task::Context<'\_>,

buf: &[u8]

) -> std::task::Poll<std::result::Result<usize, std::io::Error>> {

self.pin().poll\_write(cx, buf)

}

// Implement other required methods...

}

**Simulate Peer Interactions:**

Now we can write tests that use these mocks to simulate receiving a block from a peer. We will use mocked **transport** to simulate and receive network messages.

#[tokio::test]

async fn test\_receive\_block() {

// Create a new instance of MockTransport with the necessary mocked behaviors.

let mut transport = MockTransport::new();

// Set up expectations for the mock transport.

// The 'times' method specifies how many times this interaction is expected to occur.

// 'returning' specifies what the mock should do when the method is called.

transport.expect\_read()

.times(1) // Specify how many times this should be called

.returning(|buf| {

// Simulate receiving a block message

let message = simulate\_block\_message(); // This function needs to be defined or mocked

buf.copy\_from\_slice(&message);

Ok(message.len())

});

// Create a peer from the transport. Make sure that `create\_peer\_from\_transport` can accept a MockTransport.

let mut peer = Peer::create\_peer\_from\_transport(transport, /\* other params \*/);

// Assume we have a method to start the peer interaction and check its result.

let result = peer.start().await;

assert!(result.is\_ok());

// Verify internal state changes or other interactions

assert\_eq!(peer.state, State::Connected);

}

We can create mock blocks and transactions that simulate real network data. These would include typical transactions, edge cases like very large transactions, invalid transactions, and double spends and test the consensus rules implementations also.

**The idea behind it is to simulate a client-server transport mechanism, to test sending and receiving data.**

* **Data flow to Utreexo and Electrum**

Here we must ensure that the data from the Bitcoin node regarding transactions and blocks correctly updates the **Utreexo accumulator** and the **Electrum server** accurately retrieves data from the Bitcoin node and **Utreexo** structure and correctly serves it to the **clients**.

**Utreexo:** We verify that each block and transaction updates the utreexo accumulator correctly and utreexo proofs for transactions are valid and updated with each block added.

**How can we do this?**

**Mock Data:** Create mock blocks, transactions, and UTXOs.

**Chain State:** Initialize a mock or test chain state that integrates Utreexo.

Initialize a test blockchain state with dummy parameters:

use crate::chainstate::{ChainState, KvChainStore};

use crate::prelude::Network;

use crate::AssumeValidArg;

fn setup\_test\_chainstate() -> ChainState<KvChainStore> {

let chainstore = KvChainStore::new("/tmp/test\_chainstate").unwrap();

ChainState::new(chainstore, Network::Bitcoin, AssumeValidArg::Disabled)

}

Example test to verify **Utreexo** updates:

#[test]

fn test\_utreexo\_insertion\_and\_deletion() {

let mut chain\_state = setup\_test\_chainstate();

let new\_block = create\_test\_block\_with\_transactions();

// Implement this function to create a block with specific transactions

// Simulate processing the block

chain\_state.process\_new\_block(new\_block).unwrap();

// Assume process\_new\_block is a method that updates the Utreexo accumulator

// Verify Utreexo state

assert\_eq!(chain\_state.utreexo.contains\_unspent(...), true, "UTXO should be present after block processing");

assert\_eq!(chain\_state.utreexo.contains\_unspent(...), false, "UTXO should be removed after spending");

}

**Performance Testing:**

Given that **Utreexo’s performance** is critical we could also implement tests that measure how quickly and efficiently the **Utreexo** can process updates.

We can use **Criterion.rs** for this, it provides benchmarks with statistical confidence, offers detailed visual graphs and enables comparative analysis to track performance or regression over time.

**How can we do this ?**

**1)** Enable the benchmark feature in our project:

[[bench]]

name = "my\_benchmark"

harness = false

2) Now we create a new file in the **`benches/`** directory in the project.

**Demo Benchmark:**

use criterion::{criterion\_group, criterion\_main, Criterion};

use my\_project::{func\_a, func\_b, func\_c}; // Assuming these are in our project

fn pipeline\_benchmark(c: &mut Criterion) {

c.bench\_function("function\_pipeline", |b| {

// Criterion calls this part many times to measure the performance of functions.

b.iter(|| {

let result\_a = func\_a(/\* parameters\*/);

let result\_b = func\_b(result\_a);

func\_c(result\_b)

});

});

}

criterion\_group!(benches, pipeline\_benchmark);

criterion\_main!(benches);

This approach measures the performance of the functions as a whole.

**3) Run Benchmarks: `cargo bench`**

**4) Analyze Results:** Criterion stores the results in the **`target/criterion`** directory by default. It generates **HTML** reports that we can open in our web browser to see the detailed graphs of our benchmarks.

The report mainly includes:

* **Mean execution time.**
* **Median execution time.**
* **Variability (noise) in the execution times.**
* **Comparative graphs if previous benchmark data exists.**

**Electrum:** To test the integration of **Electrum** server with the **Bitcoin node,** and ensure that it correctly serves data to clients, we’ll want to create tests that simulate real – world integrations and data flow.

**Demo Implementations:**

**Simulating client connection and Data Request:** This test would simulate a client connecting to the Electrum server, making a request, and receiving a response. This will test the basic connectivity and functionality of the server.

#[async\_std::test]

async fn test\_client\_connection\_and\_data\_request() {

let server = ElectrumServer::new("127.0.0.1:50001", /\* other parameters\*/).await?;

// Simulate a client connecting and making a request

let client\_stream = TcpStream::connect("127.0.0.1:50001").await?;

let client = Arc::new(Client::new(1, Arc::new(client\_stream)));

// Simulate sending a request from the client

let request = Request {

id: 1,

method: "blockchain.transaction.broadcast".to\_string(),

params: vec![json!("tx\_data\_here")]

};

let response = server.handle\_client\_request(client, request).await;

// Assert response is as expected

assert!(response.is\_ok());

}

**End-to-End Data Flow Test:** This testwould ensure that when a transaction is broadcast via the node interface, it’s correctly processed and clients subscribed to relevant script hashes receive updates.

This involves the flow from **node -> server -> client.**

#[async\_std::test]

async fn test\_transaction\_broadcast\_and\_client\_notification() {

// Set up the Electrum Server, the server need to be in a known

// state to accurately assess the behavior of the code under test

// conditions.

let server = setup\_test\_server().await;

// Simulate the broadcasting of a transaction via the node

let tx = Transaction { /\* transaction data \*/ };

server.node\_interface.broadcast(tx.clone()).await.unwrap();

// Ensure the transaction is processed by the server

assert!(server.chain.broadcast(&tx).is\_ok());

// Simulate a client that is subscribed to addresses affected by the transaction

let client\_stream = TcpStream::connect("127.0.0.1:50001").await.unwrap();

let client = Arc::new(Client::new(1, Arc::new(client\_stream)));

// Client subscribes to a script hash

let script\_hash = sha256::Hash::hash(tx.output[0].script\_pubkey.as\_bytes());

server.client\_addresses.insert(script\_hash, client.clone());

// Assert that the client receives a notification about the transaction

let expected\_notification = json!({

"jsonrpc": "2.0",

"method": "blockchain.scripthash.subscribe",

"params": [script\_hash, /\* expected status \*/]

});

let notification = client.read().await;

assert\_eq!(notification, expected\_notification);

}

The final step checks whether the client receives the expected notification about the transaction. This is critical as it confirms that the **server** correctly identifies events relevant to a **client’s** subscription and communicates these events back to the **client** effectively.

**As mentors would better understand the whole architecture, I would request their help for guidance and work alongside them to identify operations of key interest.**

### 3**. Performance Testing:**

Due to intensive data and network operations involved in validating **blockchain** **transactions** and handling **multiple** **client** **requests**, performance testing is essential. It helps ensure that the system can handle expected loads, processes transactions within acceptable time frames, and scales effectively under high demand. Below are some ideas for an efficient performance testing infrastructure:

* **Load Testing:** Simulate a large number of **clients** connecting to the **Electrum** Server to request data like balance checks, transaction history and see how the system handles these requests.
* **Stress Testing:** Push the system beyond its normal operational limits to see how it performs under stress. This can include extreme numbers of transactions or rapid-fire sequences of blocks and reorgs.

Evaluate how well and quickly the system recovers from crashes or overloads, which is critical for maintaining uptime and reliability.

* **Scalability Testing:** Limit the resources **(CPU, RAM)** available to the system to see how it impacts performance metrics. This helps in understanding the scalability of the system with limited hardware capabilities.

**Linux:** We can use **`*cgroups*`** and **`*cgroups*-fs**` [Rust crate that allows manipulating and interfacing with the **cgroups tools** on Linux] to limit resources like **CPU** and **RAM**.

* **Endurance Testing:** Monitor the system under extended periods under normal load conditions to identify potential performance degradation issues, such as memory leaks, database locks, or resource saturation.
* **Concurrency Testing:** Simulate multiple clients accessing the **same** data or resource simultaneously to check how concurrent accesses affect performance. This is important for the **Electrum server** where many clients might query the same transaction or block data.
* **Latency Testing:** Measure how network delays affect the performance of data requests and response between the node and Electrum server, and between server and clients. This is especially significant for geographically distributed clients.

**How can we do this?**

1. **Setup and Initial Monitoring:**

Tools used: Prometheus, Grafana, tshark

* [**Prometheus**](https://docs.rs/prometheus/latest/prometheus/)**:** It is a monitoring and alerting toolkit used in observing software applications and infrastructure. It can collect a wide range of metrics that are essential for understanding system performance.
* [**Grafana**](https://grafana.com/docs/pyroscope/latest/configure-client/language-sdks/rust/)**:** It is an open-source analytics and monitoring solution that is widely used for visualizing time-series data. **Grafana** can graph latency data over time, providing a clear and dynamic visual representation of how network or system latencies fluctuate.
* [**Tshark**](https://www.wireshark.org/docs/man-pages/tshark.html)**:** It is the command-line version of **Wireshark,** a powerful tool used for capturing and analyzing network packets. **Tshark** can capture all the packets traveling over a network on specified interfaces and ports.

1. Setup **Prometheus** to collect metrics from the bitcoin node and Electrum server. We configure it to specifically gather data related to request handling times, system load, and network traffic.
2. Use **Grafana** to create dashboards that visualize the metrics collected by **Prometheus**.

Now we configure **Tshark** to capture traffic on the network interface that the node and server use for communication to ensure that we are monitoring the correct traffic without excessive unrelated data.

**Capture and Filter Network Traffic**

***Command to start capture:***

tshark -i eth0 -f "port 50001 or port 8333" -w /path/to/save/capture.pcap

**`-i eth0`**specifies the network interface

**`-f`** sets a capture filter (e.g. capturing traffic on typical **Bitcoin node** (**8333**) and **Electrum server** (**50001**) port)

**`-w`** writes the captured data to a file.

**Analyze traffic:**

After capturing it we can run various **Tshark** commands to analyze the traffic. For instance to read through the capture and summarize the interactions, we can use:

tshark -r /path/to/save/capture.pcap -T fields -e ip.src -e ip.dst -e tcp.analysis.ack\_rtt -E header=y -E separator=, -E quote=d

This command reads the captured file and extracts fields such as:

**Source and** **Destination** **IP Addresses:** By extracting `**ip.src`** and **`ip.dst`**, we identify the traffic flow paths. This helps in visualizing and understanding the network routes taken by data packets between clients and our server. It's vital for diagnosing inefficient routing or identifying potential points of failure and congestion in the network.

**Round-Trip Time (RTT) Analysis:** The `**tcp.analysis.act\_rtt`** field captures the acknowledgement round-trip time for **TCP packets,** which is direct measurement of network latency between source and destination. Analyzing this helps us understand the delay characteristics of the network, specifically how long it takes for data to travel back and forth within the network.

1. **Network Conditioning and Testing:**

Tools Used: WANem, NetEm

* [**WANem**](https://wanem.sourceforge.net/)**:** Wide Area Network Emulator is used specifically for testing the performance of network applications and services under various network conditions.

1. Configure **WANem** or [**NetEm**](https://man7.org/linux/man-pages/man8/tc-netem.8.html)on our **Linux** server to introduce specific network conditions such as increased latency, jitter and packet loss. These conditions simulate various realistic network environments.
2. While these tools manipulate network traffic, we must continuously monitor how these changes affect the **performance** **metrics.**
3. **Load and Stress Testing:**

Tools used: JMeter

* **JMeter:** It is a load testing tool for analyzing and measuring the performance of various services. JMeter can create heavy load by simulating multiple users or requests to the server simultaneously.

1. We will use [**JMeter**](https://jmeter.apache.org/)to simulate multiple clients accessing the **Electrum** server or sending transactions to the **Bitcoin** **node**. These tools can generate high loads and concurrent requests, testing the robustness of the system under stress.
2. We will have to develop and run custom **scripts** that automate specific **transactions** or **data** **requests** to the server. These scripts can log detailed timing information for each operation, helping pinpoint where delays occur under **load**.

**Example Scenario:**

1. **Environment Setup:** [**JMeter**](https://dev.to/hitjethva/how-to-install-apache-jmeter-on-ubuntu-20-04-2di9), Prometheus and Grafana

2. **Metrics:**

**-** Transaction throughput

- Response time for fetching data from the Electrum Server

- Memory and CPU usage

- Block processing time, particularly how quickly new blocks are processed

3. **Create JMeter Test Plan:**

We launch JMeter and create a **Test Plan,** add a **Thread Group** to simulate virtual users.

Now we configure **HTTP** requests to simulate Electrum client actions, we can do all this in the JMeter GUI, checkout the setup [here.](https://www.guru99.com/hands-on-with-jmeter-gui.html)

4. **Setup Data Inputs:**

We will use **CSV Dataset Config** in JMeter to input transaction data, block identifiers, or addresses that **virtual users** will use.

**5. Implement Specific Test Scenarios:**

**a. High Transaction Volume**

**b. Read Intensive Load from Electrum Server:** Simulate hundreds of clients fetching transaction data simultaneously.

**c. Sustained Load:** Constant load of 500 transactions per minute for several hours.

**6. Monitoring and Analysis:** Now we configure Prometheus and Grafana to scrape metrics and create dashboards.

use prometheus::{HistogramOpts, Histogram, register\_histogram};

// Metrics for processing different Electrum requests

let request\_processing\_latency = register\_histogram!(

HistogramOpts::new("electrum\_request\_processing\_latency", "Time taken to process Electrum requests (seconds)")

.buckets(vec![0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0]), // Tailor buckets to expected response times

"request\_type"

).unwrap();

// Metrics for transaction broadcast latency

let transaction\_broadcast\_latency = register\_histogram!(

HistogramOpts::new("electrum\_transaction\_broadcast\_latency", "Time taken to broadcast transactions (seconds)")

.buckets(vec![0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0]), // Tailor buckets to typical network conditions

"transaction\_type"

).unwrap();

// Metrics for block processing time

let block\_processing\_latency = register\_histogram!(

HistogramOpts::new("electrum\_block\_processing\_latency", "Time taken to process blocks (seconds)")

.buckets(vec![0.1, 0.5, 1.0, 2.0, 5.0, 10.0]), // Reflect typical block processing times

).unwrap();

We can refer how to setup **prometheus** and **grafana** for the metrics from [**fedimint-metrics.**](https://github.com/fedimint/fedimint/blob/master/fedimint-metrics/src/lib.rs)

We could also develop a mechanism to integrate all these **monitoring** **tools** and **systems** into our **CI/CD pipeline.** This allows for ongoing performance evaluation with each deployment, ensuring that new changes do not introduce new **latency** **issues** or **regress** on previous optimizations.

By following these steps, we can methodically analyze and **improve** the latency characteristics of our **Bitcoin** and **Electrum** server system, enhancing overall performance and user experience.

**I would request the help of my mentors to guide me through the process.**

### 4. **Security Testing:**

Security testing is crucial for any system, especially in cryptocurrency transactions like a **Bitcoin** node integrated with an **Electrum** server. Below are some specific ideas for security testing in our system:

1. **Input validation:**

**Fuzz testing:** We can use fuzzing techniques to input random, malformed, or unexpected data into the system to ensure it handles such inputs gracefully without crashing or exposing vulnerabilities.

**Example:**

[**Honggfuzz**](https://docs.rs/crate/honggfuzz/0.2.1)**:** A security oriented **fuzzer** used primarily to find bugs by automatically feeding unexpected or random data to the program and observing its execution for crashes, assertions, or potential vulnerabilities.

use honggfuzz::fuzz;

fn main() {

loop {

fuzz!(|data: &[u8]| {

// Attempt to decode the data into a transaction

if let Ok(tx) = Transaction::consensus\_decode(&mut &data[..]) {

// Run the verification function

let result = verify\_transaction(&tx);

// You can assert for expected outcomes based on what you consider a "valid" fuzz input

// For instance, if you want to ensure that valid decoded transactions also pass verification:

assert!(result.is\_ok(), "Verification failed for a transaction that should be valid: {:?}", result);

}

});

}

}

Here **honggfuzz** continuously feed random data to test the transaction verification function. A similar kind of fuzzing technique is used in [**fedimint**](https://github.com/fedimint/fedimint/blob/master/fuzz/src/bin/string.rs)**.**

1. **Network Security:**

**Port Scanning:** Identify open ports on the system and ensure that only necessary ports are exposed and properly secured. Helps in mapping out the **network structure**, understanding how services are **distributed across the network**, and how **traffic flows** through it.

**Tools:**

**Nmap:** The Network Mapper is a free and open-source tool for network discovery and security auditing.

**How to do Port Scanning with Nmap?**

**1) Installation: `sudo apt-get install nmap`**

**2) Basic scan:** To perform a basic scan of a target system to

- identify open ports, **`nmap [target IP or hostname]`**

- scan range of ports, **`nmap –p 1-1000 [target IP or hostname]`**

**3) Aggressive Scan:** This performs OS detection, version detection, script scanning and traceroute**: `nmap –A [target IP or hostname]`**

**Sensitive Data Exposure:** Test for exposure of sensitive data through logs, error messages or data leaks via cache or backup files.

1. **Incident Response:**

**Breach Simulation:** Conduct simulated security breach scenarios to test the effectiveness of incident response plans.

**Recovery Procedures:** Ensure that there are comprehensive and tested disaster recovery and data backup procedures to recover from security incidents without data loss.

### 4. **Regression Testing:**

Regression testing is crucial for maintaining the stability and reliability of software as it evolves. It involves re-running previously conducted tests to ensure that new changes haven’t adversely affected existing functionality. Below are some related ideas:

**1) Automated Test Suites:** We can build an automated test suite that covers all functionalities of the system. This should include unit tests, integration tests, and functional tests that can be rerun easily after every change.

We can integrate these regression tests into a CI pipeline to run automatically on every commit or merge into main branches.

**2) Cross-Version Compatibility Testing:** This testing ensures that software remains functional and consistent across different versions of itself or its dependencies.

Some key areas to consider are:

* **API Compatibility:** Ensure that the Electrum server’s API can serve clients of different versions.
* **Data Format Compatibility:** Verify that all versions can read and write data formats used across the system (e.g. Transaction formats, block data structures).
* **Protocol Compatibility:** Check that different versions can communicate using the underlying network protocols.

**How can we do this?**

**A) Identify Version Combinations:** We can create a matrix of the version combinations that need to be tested. For instance:

* Node versions: **1.0, 1.1, 1.2**
* Electrum server versions: **2.0, 2.1, 2.2**

We might end up with combinations like Node 1.0 with Electrum 2.0, Node 1.1 with Electrum 2.1, etc.

**B) Setup Testing Environment:** We must configure a test environment where we can deploy different versions of our Bitcoin node and Electrum server. This might involve:

**Virtual Machines or Containers:** Use VMs or containerized environments (like Docker) to isolate and run multiple versions simultaneously without interference.

**Automated Deployment Tools:** Use scripts or **CI/CD** pipelines to deploy different versions quickly.

**C) Create Test Cases:** Develop test cases that cover typical user interactions and edge cases across versions. These should include:

* **Transaction Processing:** Send transactions from a client of one version to a server of another version and vice versa.
* **Data Retrieval:** Request data from the server using clients of different versions to ensure responses are correctly understood and handled.
* **Error Handling:** Test how systems handle errors when interacting with incompatible versions or unsupported features.

## **Timeline**

**Program Kick-Off and Onboarding Period – May 8 to May 15, 2024**

1. Familiarize me completely with the project’s functionality and architecture.
2. Pair with my mentor and plan on achieving the project’s milestones.
3. Make contributions to the project and examine existing contributor patterns.
4. Attend talks and seminars on various Bitcoin-related topics.

**Coding Period – May 15 to Aug 15, 2024**

**May 15 – May 22:**

1. Set up configuration for performing unit testing.
2. Identify critical operations with consideration for mentor and community.
3. Write down unit tests for all over the codebase.

**May 22 – May 29:**

1. Finish up with the unit testing for florestad and libfloresta.
2. Having identified bugs, list all of them down.

**May 29 – June 5:**

1. Start identifying the critical aspects for the codebase for integration testing and setting the test environment.
2. Write down some basic integration tests.
3. Discuss with my mentor regarding complex integration tests.

**June 5 – June 12:**

1. Progress with the integration tests.

**June 12 – June 21:**

1. Finish up with the integration testing.
2. Documenting the work done in unit and integration tests, for easy understanding for newcomers.
3. Listing down the bugs, if any found.
4. Finish up with Security and Regression testing.
5. Integrating the testing infrastructure with CI/CD pipe

**June 21 – June 30:**

1. Integrate the present testing infrastructure with the CI/CD pipeline.
2. Research about starting up with the performance testing setup.
3. Setting up the tools which are going to be used: Prometheus, Grafana, tshark, cgroups, WANem, JMeter.

**Mid-Term Evaluations – July 1 – July 5:**

1. **Mentors submit mid-term evaluations of their mentees.**

**July 1 – July 17:**

1. Analyze the data recovered from the monitoring tools and look out for latency operations.
2. Finish up with stress, scalability, endurance and concurrency testing.
3. Document the work and list down the findings.

**July 17 – July 31:**

1. Setting up configuration for latency and benchmark testing (Criterion.rs)
2. Setup network conditioning tests.
3. Having identified latency operations, list all of them down and separate them out.
4. Finishing up with the JMeter load testing.
5. Catching up with the pending tasks.

**July 31 – August 7:**

line.

**August 7 – August 15:**

1. Documenting the testing infrastructure.
2. Gather a review of the work done till now from the community.
3. Writing down a blog on what I did this summer with Floresta and sharing it.
4. Completing pending tasks and moving towards final evaluation.

**Post SoB:**

1. I will continue contributing to Floresta by adding more features and helping maintain the repository after SoB ends.

**Final Evaluations – August 15 – August 30, 2024**

1. **Mentors do a final review code and determine if the students have completed their Summer of Bitcoin project.**