SoB 2024 Proposal

Harsh Raj
harsh.raj.cd.ece21@iitbhu.ac.in
Bihar, India
Timezone: Indian Standard Time (+5.30 UTC)

VLS Recovery Framework

Enhancing Recovery & Fault Tolerance in Validating Lightning Signer

Personal Details and Contact Information

Hi! I am Harsh Raj, a pre-final year student from the *Indian Institute of Technology (BHU) Varanasi, India*. I am an avid open-source enthusiast who loves making (and breaking) software.

You might better recognize me as **Harsh1s**, as I have contributed a bit to VLS in the past few months. Here are my details and contact information:

- **Email Address:** harsh.raj.cd.ece21@itbhu.ac.in
- **Major:** Electronics and Communication Engineering
- **Degree:** Bachelor of Technology (B. Tech.)
- **Matrix:** @harsh1s:matrix.org
- **Time-zone:** Indian Standard Time (+5:30 UTC)
- **LinkedIn:** https://www.linkedin.com/in/harsh1s
- **Resume:** https://aquamarine-elenore-41.tiiny.site
- **Postal Address:** Postal Park Road no. 2, Kankarbagh, Patna, Bihar (800001), India
- **Phone:** (+91) 85446 33455

Synopsis

The Validating Lightning Signer (VLS) project is instrumental in improving the security of the Lightning Network by isolating private keys and policy enforcement from potentially compromised Lightning nodes. This Summer of Bitcoin project seeks to solidify the robustness of VLS by developing comprehensive recovery mechanisms, detailed logging, and rigorous testing procedures. The envisioned framework will enable users to recover their funds and maintain trust in the system, even in the event of unexpected failures.

Benefits to the Community

The successful implementation of this recovery framework offers significant benefits to the entire Lightning Network community:

- **Enhanced Trust and Adoption:** Users gain greater confidence in Lightning knowing that comprehensive recovery exists for their funds. This removes a major barrier to adoption, particularly for larger holdings and business use cases.
- **Improved Node Reliability:** Lightning node operators will be empowered to run larger, more well-funded nodes, knowing that VLS minimizes risks associated with unexpected failures. This increases the overall capacity and efficiency of the network.
- **Strengthened Ecosystem:** A robust VLS solidifies its position as a core security component, encouraging collaboration between VLS, hardware manufacturers, and various Lightning node implementations. This fuels innovation in securing Lightning transactions.

- **Reduced Support Burden:** Clear documentation, guidelines, and recovery tools decrease user reliance on support channels for failure scenarios. This frees up developer resources and fosters self-reliance within the community.
- **Contributions to Open-Source:** The project will produce thoroughly documented open-source code, best practices, and test suites. This benefits the overall health of the VLS project and provides a valuable knowledge base for the wider Lightning development community.

Broader Implications

Beyond the immediate improvements to VLS, this project showcases principles and techniques applicable to secure other critical components within the Lightning Network ecosystem:

- **Robust State Management:** The state backup and restoration patterns developed here can be valuable references for other projects requiring reliable state management in distributed systems.
- **Failure Testing:** Our methodology for simulating failures and meticulously testing recovery scenarios provides a blueprint that other Lightning projects can adapt.
- **Security through Specialization:** The focus of VLS on custody and policy enforcement underscores the value of component isolation to improve overall security in complex systems.

Current Status of the Project

1. **Core Functionality:** The VLS project has a successful implementation of the core functionality of separating Lightning private keys and security rule validation from the Lightning node, improving the security of the Bitcoin Lightning network. This separation enhances security by reducing the attack surface and mitigating the risk of compromising the entire Lightning node in case of a security breach.
2. **Partial State Backups:** VLS supports some state persistence, providing a foundation for recovery. However, the completeness and automation of this process is limited. Users may need in-depth knowledge of Lightning protocols and VLS internals to devise their own recovery strategies effectively.
3. **Rudimentary HTLC Handling:** While the concept of HTLC sweeps exists, implementation is potentially incomplete or lacks rigorous testing under diverse failure conditions, specifically covering both incoming and outgoing HTLCs.
4. **Limited Justice Transaction Support:** The ability to detect on-chain commitment breaches and construct the corresponding justice transactions to recover funds require enhancement.
5. **Basic Recovery Mechanisms:** While the project has basic error recovery mechanisms in place, it does not have robust mechanisms in place to handle various failure scenarios effectively. Restoring from partial backups and constructing justice transactions might involve numerous manual steps prone to errors. Also, in the event of a node or signer crash, data loss, or power outage, the system may experience downtime or loss of critical data, potentially jeopardizing the integrity of Lightning transactions.
6. **Fault Tolerance:** The current implementation of VLS does not incorporate advanced fault tolerance strategies to minimize the impact of errors. As a result, the system may be vulnerable to disruptions caused by hardware failures, network issues or other failures.
7. **Error Analysis and Logging:** While VLS does include logging mechanisms to capture certain error information, the current logging capabilities are limited in scope and may not provide sufficient detail for diagnosing and resolving issues promptly. Improving error analysis and logging will facilitate quicker diagnosis and resolution of issues, improving overall system reliability and stability.

8. **Testing and Breaking:** Unit tests might cover core VLS logic. However, there is a need for more comprehensive testing, including stress testing and simulated failure scenarios, to identify potential vulnerabilities and weaknesses in the recovery mechanisms.
9. **Documentation and Guidelines:** Inadequate documentation and a lack of thorough failure testing lead to uncertainty for users about whether funds can be reliably recovered in different scenarios. Enhancing the documentation to provide clear guidance and instructions will be essential for users to understand and implement the recovery procedures effectively.

Goals

1. Comprehensive Recovery Framework

- **State Backup and Restoration:**
    - **Robust State Identification:** Define and track all critical VLS state elements required for recovery (channel data, policies, preimages, etc.).
    - **Secure Backup Mechanisms:** Design secure local and optional cloud-based state backup strategies, emphasizing encryption and user control over keys. Design state serialization and validation logic for seamless restoration to a new VLS instance.
    - **Restoration Procedures:** Develop step-by-step procedures for restoring VLS state from backups, handling both partial and complete data loss scenarios.
- **HTLC Handling:**
    - **HTLC Identification:**
        - **In-Memory Tracking:** Maintain an active list of pending HTLCs, including:
            - HTLC amount, timeout/expiry block height, preimage, associated channel, and direction (incoming vs. outgoing).
        - **State Persistence:** Serialize HTLC data as part of the VLS core state for backup and recovery.
    - **Sweep Logic (Outgoing HTLCs):**
        - During a shutdown, locate HTLCs for which VLS holds the preimage.
        - If the timeout is approaching/expired, broadcast sweep transactions to reclaim funds based on current chain state.
    - **Sweep Logic (Incoming HTLCs):**
        - Identify HTLCs where VLS is intended to *receive* the funds.
        - If the HTLC hasn't been resolved normally after restoration, broadcast a sweep transaction if the timeout is approaching or has passed.
- **Justice Transaction Support:**
    - **State Comparison:** VLS will need to compare restored channel states with the latest commitment transaction on-chain after a failure.
    - **Breach Detection:** If a past commitment state was revoked, construct the necessary justice transaction. This requires knowing the revocation secrets for older states.
    - **Transaction Broadcast:** Carefully manage the broadcast of justice transactions, potentially prioritizing based on value at stake.

2. Thorough Testing & Validation

- **Test Environment:** Construct a robust test environment, including containerized Lightning networks (mainnet & testnet) and scripting capabilities to induce controlled failures.

- **Failure Scenario Coverage:** Create comprehensive test suites covering:
    - Node/signer crashes at various points during the channel lifecycle
    - Data loss and corruption (simulating disk failures, etc.)
    - Power outages with varying degrees of state preservation.
    - Cooperative and uncooperative channel closure scenarios.
    - Recovery procedures with varying channel states.
- **Stress Testing:** Implement stress testing to assess the system's performance under extreme conditions and ensure that it can withstand high loads and unexpected events.
- **Automated Test Suite:** Develop automated test cases for recovery scenarios, facilitating regression testing as VLS evolves.

3. Documentation and Guidelines:

- **Recovery Processes Documentation:** Create comprehensive documentation detailing the recovery processes. This should include user-friendly, step-by-step guides tailored to different levels of technical expertise, ensuring users can easily understand and execute the necessary steps in case of failures.
- **Technical Recovery Procedures:** Document every step for backup, restoration, HTLC sweeps, and other relevant procedures. This should also cover recovery procedures in different scenarios, providing clear instructions and guidelines for implementing and configuring the recovery mechanisms.
- **Best Practices:** Develop a set of best practices to help users prevent data loss and ensure transaction integrity. This should include security guidelines for minimizing risk, such as implementing redundancy measures, and recommendations for system configuration and backup strategies for maximizing the likelihood of successful recovery.

4. Enhanced Fault Tolerance

- Implement improved fault tolerance strategies within the VLS system to minimize the impact of errors and failures.
- Enhance the system's ability to detect and recover from faults automatically, reducing the need for manual intervention and improving overall system reliability.
- **Fault Tolerance Strategies:**
    - **Redundancy Options:**
        - **Full Replication:** Potentially costly, but a secondary VLS instance mirroring state offers quick failover.
        - **Partial Replication:** Only replicate critical state elements for reduced overhead (open channels list, recent commitments, but not every old revocation secret).
        - **Checksums:** Integrity checks on stored state help detect early corruption.
    - **Graceful Shutdown:**
        - **RPC Hooks:** Integrate VLS with the Lightning node to receive controlled shutdown notifications.
        - **Commit Pending Changes:** Commit in-flight transactions to known states (even if not fully settled yet) before stopping the VLS.
        - **Sweep Initiation** Begin HTLC sweeps within a timeout window before the complete shutdown.
    - **Transaction Safety:**
        - **Timeout Awareness:** Account for relative lock times & expiry blocks in sweeps and justice transactions to minimize the risk of those transactions becoming invalid themselves.

5. Error Analysis and Logging

- Implement advanced error analysis tools to facilitate quick diagnosis and resolution of issues, helping to minimize downtime and data loss.
- Expand the VLS logging system to include:
    - Precise timestamps
    - Granular error codes
    - Details of transactions and channel states involved in the error

Stretch Goals

- **Advanced Replication:** Investigate multi-signer or multi-device replication for ultra-high availability, potentially incorporating consensus protocols for state updates.
- **Automated Recovery:** Explore whether certain recovery actions (HTLC sweeps, limited justice transactions) could be partially automated, with safeguards, to reduce the burden on users during failures.
- **Disaster Recovery Testing:** Create a specialized test suite simulating large-scale disasters (e.g., multiple node failures) to assess VLS behavior in extreme scenarios.

Out-of-Scope Goals

- **On-Chain Analysis Tooling:** Developing full-fledged blockchain explorers or complex analysis tools for commitment states is outside the scope. (However, integrating with existing tools could be considered).
- **Non-Lightning Recovery:** Recovering funds outside of the VLS/Lightning context (lost seed phrases, general wallet issues) is not a focus.
- **Performance Optimizations:** While improved code efficiency is always good, this project prioritizes recovery correctness rather than raw performance tuning.

Deliverables

1. Core VLS Code Enhancements

- **State Management:**
    - Expanded state tracking mechanisms to encompass recovery-specific data.
    - Secure local (and potentially cloud-based) state backup and restoration functionality with appropriate encryption and access control.
    - Implementation of fault tolerance strategies (replication, failover mechanisms, automatic error detection and recovery, and graceful shutdown handling).
- **HTLC Logic:**
    - HTLC identification and tracking system integrated with the Lightning node.
    - Sweep transaction logic (both incoming and outgoing HTLCs) during recovery.
- **Justice Transaction Logic:**
    - Reliable channel commitment state tracking.
    - Capabilities to construct and broadcast on-chain justice transactions in response to breaches.
- **Enhanced Logging:**
    - Granular logging for error scenarios, including timestamps, component identifiers, transaction details, and other relevant debugging information.

2. Testing Infrastructure and Suites

- **Containerized Test Environment:** A reproducible test environment with containerized Lightning nodes and VLS instances, allowing for controlled failure injection.
- **Failure Scenario Test Suites:** Comprehensive test cases covering:
    - Node/signer crashes at critical points
    - Varied data loss and corruption simulations
    - Power outage scenarios
    - Cooperative and uncooperative channel closure cases
- **Automated Test Suite:** A readily extensible automated testing framework to ensure recovery feature robustness and facilitate regression testing.

3. Documentation and Guidance

- **Technical Recovery Documentation:**
    - In-depth description of VLS state backups and restoration procedures.
    - Technical explanations of HTLC sweep and justice transaction mechanisms.
    - Best practices for VLS configuration and deployment to maximize recovery chances.
- **User-Friendly Recovery Guides:**
    - Step-by-step instructions for various recovery scenarios, tailored to different levels of technical expertise.
    - Troubleshooting guides for common issues encountered during the process.
- **Log Analysis Tools:** Scripts or simple tools to aid in log parsing and troubleshooting.

Expected Results

- **Robust Fund Recovery:** VLS users gain significantly increased confidence that their Lightning funds are recoverable in the event of unexpected failures, data loss, or node/signer crashes. This removes a significant barrier to adoption of Lightning for larger holdings and business use cases.
- **Strengthened Lightning Node Reliability:** Lightning node operators can run larger, better-funded nodes knowing that comprehensive recovery mechanisms exist within VLS, minimizing risks associated with potential malfunctions. The overall Lightning Network capacity and efficiency could increase as a result.
- **Improved User Experience:** Clear recovery instructions and well-designed tools empower both technical and non-technical users to navigate recovery situations effectively. This reduces support burdens on developers and fosters self-reliance within the community.
- **Reduced Support Overhead:** The combination of robust recovery mechanisms, detailed documentation, and automated testing decreases the reliance of Lightning users on direct developer support for failure scenarios. This frees up developer resources to focus on innovation.
- **Contributions to Open-Source Resilience:** The project's code enhancements, test suites, and knowledge sharing directly improve the security and robustness of the VLS project as well as benefit the broader Lightning development community.
- **Quantifiable Success Rate:** Strive to define a target success rate for recovery across a wide range of tested failure scenarios (e.g., "VLS achieves 90%+ fund recovery rate in tested node crash scenarios").

Approach

1. Comprehensive Recovery Framework

- **State Backup and Restoration:**

- **Robust State Identification:**
    1. **Critical Channel Data:**
        - Channel IDs (`ChannelId`)
        - Channel setup parameters (`ChannelSetup`, `ChannelPublicKeys`)
        - Funding outpoint (`OutPoint`)
        - Commitment transaction data, including per-commitment secrets and signatures
        - HTLC state (offered, received, preimages)
        - Justice transaction data if necessary

Code Example:

```rust
#[derive(Serialize, Deserialize, Debug)]
struct ChannelState {
    channel_id: ChannelId,
    setup: ChannelSetup,
    funding_outpoint: OutPoint,
    commitment_txs: Vec<CommitmentTransaction>,
    htlcs: HtlcState, // Structure containing HTLC details
    // ... (add justice transaction state if applicable)
```

    2. **VLS-Specific Policies:**
        - Fee policies (base fee, fee rate)
        - HTLC timeout values
        - Allowlist/Denylist configurations

Code Example:

```rust
#[derive(Serialize, Deserialize, Debug)]
struct VLSPolicies {
    fee_base: u32,
    fee_rate: u32,
    htlc_timeout_default: u32,
    allowlist: Vec<String>,
    // ...
}
```

    3. **Preimages:**
        - Store preimages for any fulfilled HTLCs.
        - Utilize a secure storage mechanism, potentially with user-provided encryption keys.

Code Example (Conceptual):

```rust
#[derive(Serialize, Deserialize, Debug)]
struct PreimageData {
    payment_hash: PaymentHash,
    preimage: [u8; 32],
}


let encrypted_preimages = encrypt(serialize(preimage_data), user_key);
store_securely(encrypted_preimages);
```

- **Secure Backup Mechanisms:**
    1. **Local Backups:**
        - Implement a file-based backup system.
        - Serialize state (`ChannelState`, `VLSPolicies`, preimages) using a robust format (e.g., JSON, CBOR).
        - Encrypt serialized data with a user-provided key or a key derived from a passphrase.
        - Allow configuration of backup frequency and location.

Code Example:

```rust
use serde::{Serialize, Deserialize};
use lightning::ln::chan_utils::ChannelPublicKeys;
use lightning::ln::{PaymentHash, ChannelId};
use lightning::{ln, HtlcState};
use aes_gcm::aead::{Aead, NewAead, generic_array::{GenericArray, typenum::U24}};
use aes_gcm::{Aes256Gcm, Key, Nonce};

// ... (Existing Structs like ChannelState, VLSPolicies, PreimageData)

#[derive(Debug)]
enum Error {
    SerializationError(serde_json::Error),
    EncryptionError(aes_gcm::Error),
    IOError(std::io::Error),
}

fn backup_state(state: &ChannelState, policies: &VLSPolicies, preimages: &[PreimageData], key: &
[u8]) -> Result<(), Error> {
    let serialized_data = serde_json::to_string(&(&state, &policies, &preimages))
        .map_err(Error::SerializationError)?;

    let cipher = Aes256Gcm::new(Key::from_slice(key));
    let nonce = Nonce::from_slice(b"unique nonce"); // Ensure unique nonce per backup

    let ciphertext = cipher.encrypt(nonce, serialized_data.as_bytes())
        .map_err(Error::EncryptionError)?;

    std::fs::write("backup.dat", ciphertext)
        .map_err(Error::IOError)?;
    Ok(())
}
```

2. **Cloud-Based Backups (Optional):**
    - Provide the option to sync encrypted backup files to cloud storage providers (Dropbox, Google Drive, etc.).
    - Use the cloud provider's OAuth mechanisms for secure authorization.
    - Emphasize the importance of user control over encryption keys.

Code Example (Conceptual):

```rust
use oauth2::{basic::BasicClient, AuthUrl, ClientId, ClientSecret, TokenResponse, TokenUrl};

// ...

#[derive(Debug)]
enum CloudBackupError {
    // ... Add specific cloud backup and oauth related errors here
}

fn cloud_backup(encrypted_data: &[u8], cloud_provider: CloudProvider) -> Result<(),
CloudBackupError> {
    let client = BasicClient::new(
        ClientId::new("your_client_id".to_string()),
        Some(ClientSecret::new("your_client_secret".to_string())),
        AuthUrl::new("https://cloud_provider/oauth/authorize".to_string())?,
        Some(TokenUrl::new("https://cloud_provider/oauth/token".to_string())?)
    );

    // ... Handle OAuth flow to obtain an access token ...

    let request = cloud_provider.upload_request("vls_backup.dat", encrypted_data, &access_token);
    request.send()?; // Or similar method depending on the cloud library
    Ok(())
}
```

- **Restoration Procedures:**
    1. **Restore from Backup:**
        - Guide users through selecting a backup file.
        - Prompt users for a decryption key/passphrase.
        - Decrypt and deserialize backed-up state data.
        - Perform thorough validation checks on restored state to ensure integrity before use.

Code Example:

```rust
// ... (Error enum as before)

fn restore_state(backup_file: &str, key: &[u8]) -> Result<(ChannelState, VLSPolicies,
Vec<PreimageData>), Error> {
    let ciphertext = std::fs::read(backup_file)
        .map_err(Error::IOError)?;

    let cipher = Aes256Gcm::new(Key::from_slice(key));
    let nonce = Nonce::from_slice(b"unique nonce"); // Retrieve nonce stored with backup
    let plaintext = cipher.decrypt(nonce, ciphertext.as_ref())
        .map_err(Error::EncryptionError)?;

    let (state, policies, preimages): (ChannelState, VLSPolicies, Vec<PreimageData>) =
        serde_json::from_slice(&plaintext)
        .map_err(Error::SerializationError)?;

    validate_restored_state(&state, &policies, &preimages)?; // Extremely crucial!
    Ok((state, policies, preimages))
}
```

- **HTLC Handling:**
    - **HTLC Identification:**
        - **In-Memory Tracking:**When a new HTLC is either offered or received by VLS, a new `PendingHtlc` structure is created. This structure will hold all the crucial data about the HTLC:
            - The HTLC's amount (in millisatoshis).
            - The block height at which the HTLC expires.
            - The preimage of the HTLC (if VLS is the sender).
            - The payment hash identifying the HTLC.
            - The ID of the channel the HTLC belongs to.
            - Whether the HTLC is incoming or outgoing.
        - **State Persistence:** Whenever VLS's state is backed up, all the `PendingHtlc` structures are serialized along with the rest of VLS state. This is *crucial* for recovery, as it allows VLS to "remember" about active HTLCs upon restoration.

Code Example (Conceptual):

```rust
use lightning::ln::{PaymentHash, PaymentPreimage};

#[derive(Serialize, Deserialize, Debug, Clone)]
struct PendingHtlc {
    amount_msat: u64,
    expiry_block: u32,
    preimage: Option<PaymentPreimage>, // For outgoing HTLCs
    payment_hash: PaymentHash,
    channel_id: ChannelId,
    direction: HtlcDirection,
}

#[derive(Serialize, Deserialize, Debug, Clone)]
enum HtlcDirection {
    Incoming,
    Outgoing,
}

// Modify your VLS state representation to include HTLCs
struct VLSState {
    // ... other state
    pending_htlcs: Vec<PendingHtlc>
}
```

- **Sweep Logic (Outgoing HTLCs):**
  - **VLS Shutdown or Restoration:** When VLS shuts down or is restored from a backup, it goes through all the `PendingHtlc` structures for outgoing HTLCs. VLS checks if the preimage is known and if the expiry time is approaching.
  - **Sweeping Imminent:** If VLS has the preimage and the HTLC is close to expiring (considering a safety margin to ensure HTLCs aren't swept prematurely due to minor clock discrepancies or network delays), it follows these steps:
    - **Construct Sweep Transaction:** Using the channel state and the preimage, VLS builds a special transaction to claim the funds of the HTLC.
    - **Fee & Broadcast:** Appropriate fees are calculated and the transaction is broadcast to the Bitcoin network.

Example Code (Conceptual):

```
use lightning::util::ser::Writeable;

// ... within your VLS shutdown or state restoration logic
fn sweep_outgoing_htlcs(vls_state: &VLSState, node: &Node) { // Assume access to a Lightning Node
    let current_block_height = node.get_chain_tip().height;

    for htlc in &vls_state.pending_htlcs {
        if htlc.direction == HtlcDirection::Outgoing && htlc.preimage.is_some() &&
           htlc.expiry_block <= current_block_height + SAFETY_MARGIN { // Add a reasonable safety
margin

            let preimage = htlc.preimage.as_ref().unwrap();

            // Construct the sweep transaction using preimage and channel state
            let sweep_tx = build_htlc_sweep_tx(&htlc.channel_id, &htlc.payment_hash, preimage,
...);

            // ... logic to add fees, finalize, and broadcast transaction (using your node
interface)
        }
    }
}
```

- **Sweep Logic (Incoming HTLCs):**
  - **Monitoring After Restoration:** After restoration, VLS periodically checks for incoming HTLCs that are close to expiring. If an incoming HTLC is about to expire and hasn't been resolved normally, VLS does the following:
    - **Construct Sweep Transaction:** VLS again uses the known channel state to build a transaction to claim the funds of the HTLC.
    - **Fee & Broadcast:** The necessary fees are added and the transaction is broadcast.

Example Code (Conceptual):

```rust
// ... within your VLS restoration or monitoring logic
fn sweep_incoming_htlcs(vls_state: &VLSState, node: &Node) {
    let current_block_height = node.get_chain_tip().height;

    for htlc in &vls_state.pending_htlcs {
        if htlc.direction == HtlcDirection::Incoming &&
           htlc.expiry_block <= current_block_height + SAFETY_MARGIN {

            // Construct the sweep transaction using channel state
            let sweep_tx = build_htlc_timeout_sweep_tx(&htlc.channel_id, ...);

            // ... logic to add fees, finalize, and broadcast transaction
        }
    }
}
```

- **Justice Transaction Support:**
    - **State Comparison:**
        - **Upon Restoration:** When VLS is restarted or restored from a backup, it needs to determine if any breach has occurred on its channels while it was offline.
        - **Channel State vs. Chain:** VLS does this by comparing its stored channel states with the latest commitment transaction recorded on the Bitcoin blockchain for each channel.
        - **Finding a Breach:** If VLS detects a mismatch where an older commitment transaction (one not known as the latest to VLS) was broadcast, it signifies a potential breach attempt.

Example Code (Conceptual):

```rust
use lightning::ln::chan_utils::{derive_private_revocation_key, ChannelPublicKeys};
use lightning::ln::{PaymentHash, ChannelId};

// ... Your existing state structures
struct ChannelState {
    // ...
    commitment_txs: Vec<CommitmentTransaction>,
    revocation_secrets: Vec<Vec<u8>>, // One vector of secrets per commitment
}

// ... Logic within restoration or monitoring
fn compare_restored_state(vls_state: &VLSState, node: &Node, channel_id: &ChannelId) {
    let chain_commitment_tx = node.get_channel_commitment(channel_id); // Retrieve most recent
on-chain

    for (index, local_commitment) in vls_state.commitment_txs.iter().enumerate() {
        if !local_commitment.is_counterparty_broadcaster &&
            chain_commitment_tx.txid() == local_commitment.txid() {

            // Potentially revoked commitment state found!
            let revocation_secret = &vls_state.revocation_secrets[index];
            try_construct_justice_tx(&local_commitment, revocation_secret, ...);
            break;
        }
    }
}
```

- **Breach Detection:**
  - **Revocation Secrets:** At the heart of justice transaction creation is the revocation secret – a piece of data VLS knows but the cheating counterparty does not. This secret was generated for each commitment transaction.
  - **Unlocking the Punishment:** Using the revocation secret associated with the breached commitment transaction, VLS can derive special keys that allow it to construct transactions claiming the funds the counterparty tried to steal.
  - **Justice Transaction Types:** There are different types of justice transactions VLS might need to construct:
    - Recovering funds from offered HTLCs the counterparty tried to cheat out of.
    - Claiming funds from received HTLCs that the counterparty unfairly revoked.
    - Other potential breach scenarios VLS supports.

Example Code (Conceptual):

```
// ...
fn try_construct_justice_tx(commitment_tx: &CommitmentTransaction, revocation_secret: &[u8],
channel_keys: &ChannelPublicKeys, ...) -> Option<Transaction> {
    let revocation_pubkey = commitment_tx.setup.counterparty_points.revocation_basepoint;

    // ... Logic to derive per-commitment point and revocation key using the secret

    if commitment_tx.is_counterparty_broadcaster {
        // Attempt to construct offered HTLC justice transactions ...
        for offered_htlc in commitment_tx.offered_htlcs {
            if let Some(tx) = construct_offered_htlc_justice_tx(&offered_htlc,
&revocation_pubkey, channel_keys, ...) {
                return Some(tx);
            }
        }

        // Similarly, construct received HTLC justice transactions ...
    }

    // ... Logic to construct other justice transaction types depending on breach

    None // If no applicable justice transactions can be built
}
```

- **Transaction Broadcast:**
  - **Reclaiming Funds:** Once VLS constructs the necessary justice transactions, it broadcasts them to the Bitcoin network. This punishes the cheating counterparty and allows VLS to reclaim its rightful funds.
  - **Potential Prioritization:** In situations where multiple justice transactions are needed, VLS might prioritize broadcasting those that protect larger amounts of funds at risk.

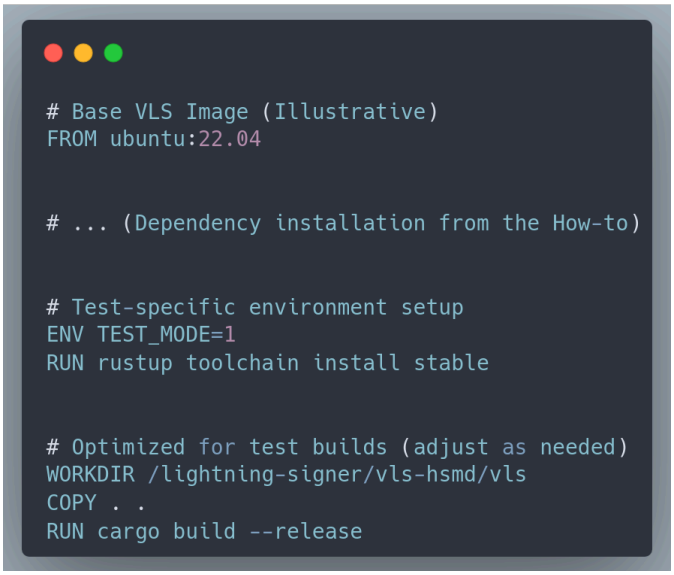
Example Code (Conceptual):

```
// ...
fn broadcast_justice_transactions(justice_txs: Vec<Transaction>, node: &Node) {
    // Consider sorting transactions by potential value at risk

    for tx in justice_txs {
        node.broadcast_transaction(&tx).expect("Failed to broadcast justice transaction");
    }
}
```

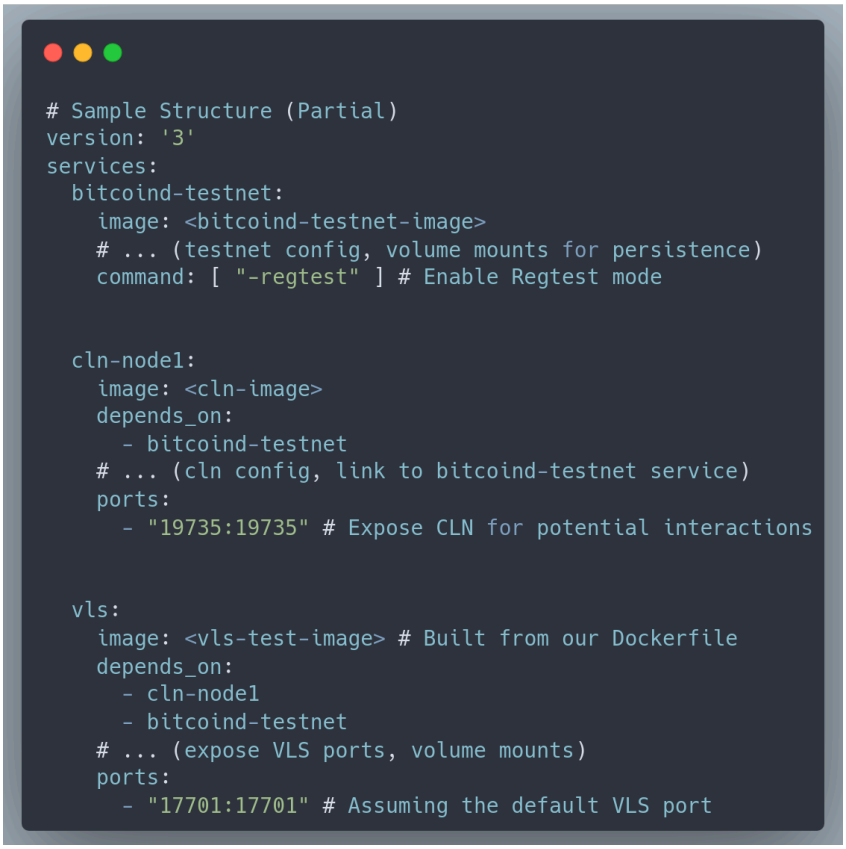
```
use lightning::ln::chan_utils::{derive_private_revocation_key, ChannelPublicKeys};
use lightning::ln::{PaymentHash, PaymentPreimage, ChannelId};
use lightning::util::ser::Writeable;

// ...(rest of the code)

fn construct_offered_htlc_justice_tx(offered_htlc: &OfferedHtlcInfo2, revocation_pubkey:
&PublicKey, channel_keys: &ChannelPublicKeys, node: &Node, ...) -> Option<Transaction> {
    let per_commitment_point = channel_keys.per_commitment_point(offered_htlc.cltv_expiry - 1);
    let revocation_basepoint = channel_keys.revocation_basepoint();
    let revocation_key = derive_private_revocation_key(&per_commitment_point,
&revocation_basepoint);

    // Construct the public key needed in the justice output script
    let delayed_payment_pubkey = channel_keys.delayed_payment_basepoint();
    let justice_pubkey =
        revocation_key.add_to_public_key(&delayed_payment_pubkey).serialize();

    // Find the offered HTLC output in the commitment transaction
    let htlc_output = commitment_tx.outputs
        .iter()
        .find(|output| {
            output.value == offered_htlc.value_sat &&
            offered_htlc.cltv_expiry == htlc_output.locktime && // Naive check for simplicity
            // ... (Potentially a more robust check for the expected output script)
        })?;

    // Build the justice transaction
    let justice_tx = Transaction {
        version: 2,
        input: vec![OutPoint {
            txid: commitment_tx.txid(),
            vout: htlc_output.index as u32
        }],
        output: vec![TxOut {
            value: offered_htlc.value_sat, // Subtract fees potentially
            script_pubkey: get_htlc_timeout_script(&justice_pubkey, offered_htlc.cltv_expiry)
        }],
        lock_time: 0,
    };

    // ... Logic to sign the transaction using VLS channel keys

    Some(justice_tx)
}
```

2. Thorough Testing & Validation

- **Test Environment:**
  - **Containerization (Docker)**
    - **Rationale:** Containerization ensures reproducibility, allows scaling, and provides isolation. Each core component (VLS, CLN nodes, bitcoind, TXOOD) would run in separate containers.
    - **Dockerfiles:**
      - Create a base container image including dependencies and a common network interface.
      - Create specialized Dockerfiles for each component:



```
# Base VLS Image (Illustrative)
FROM ubuntu:22.04


# ... (Dependency installation from the How-to)


# Test-specific environment setup
ENV TEST_MODE=1
RUN rustup toolchain install stable


# Optimized for test builds (adjust as needed)
WORKDIR /lightning-signer/vls-hsmd/vls
COPY . .
RUN cargo build --release
```



```
# Sample Structure (Partial)
version: '3'
services:
  bitcoind-testnet:
    image: <bitcoind-testnet-image>
    # ... (testnet config, volume mounts for persistence)
    command: [ "-regtest" ] # Enable Regtest mode


  cln-node1:
    image: <cln-image>
    depends_on:
      - bitcoind-testnet
    # ... (cln config, link to bitcoind-testnet service)
    ports:
      - "19735:19735" # Expose CLN for potential interactions


  vls:
    image: <vls-test-image> # Built from our Dockerfile
    depends_on:
      - cln-node1
      - bitcoind-testnet
    # ... (expose VLS ports, volume mounts)
    ports:
      - "17701:17701" # Assuming the default VLS port
```

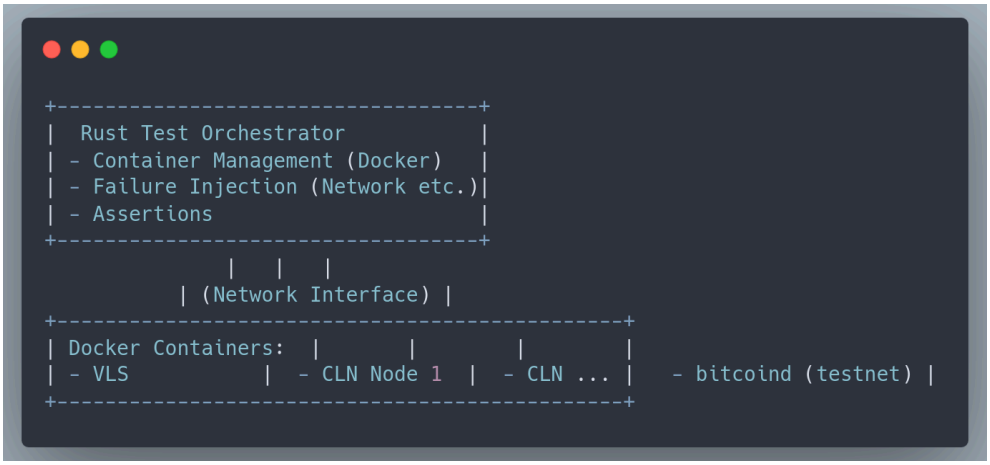
        - **Docker Compose:** Use docker-compose to orchestrate and configure the multi-container setup, defining volumes, networks, and dependencies between services.
    - **Lightning Network Setup**
        - **Network Topology:** Design various network topologies (e.g., simple 2-node, multi-node setups, isolated sub-networks) for different test cases.
        - **bitcoind:**
            - Configure multiple bitcoind instances, one for mainnet (if needed) and one for testnet, exposed to the appropriate containers.
            - Utilize Regtest mode for bitcoind to provide flexibility in blockchain manipulation.
        - **CLN Nodes:** Use CLN's configuration to create and connect nodes, ensuring alignment with the network topology within the containerized system.
    - **Scripting and Failure Injection**
        - **Rationale:** Prioritize Rust for performance, type safety, and potential integration with VLS internals.
            - **Crates:**
                - `libtest` for built-in test structuring.
                - `docker-api` for container interaction.
                - `tokio` for asynchronous operations (if needed).
                - Crates for network manipulation or specific failure simulation.
        - **Scenario Definition:**
            - **Channel Establishment:** Functions to open channels between designated CLN nodes with varying capacities and configurations.
            - **HTLC Flow:** Create HTLCs, simulate successful resolutions, timeouts, and preimage reveals.
            - **Failure Injection Points:**
                - Crashing containers to simulate node failures.
                - Network disruptions (iptables or network emulation tools).
                - Inducing data corruption in VLS's storage.

Example Code (Conceptual):

```rust
use docker_api::{Docker, Container};
use std::time::Duration;
use anyhow::Result;

// ... Test setup (Error handling omitted for brevity)

async fn channel_setup(docker: &Docker, cln_container: &Container) -> Result<()> {
    // ... Rest of the code ...
    Ok(())
}

async fn inject_cln_node_crash(docker: &Docker, cln_container: &Container) -> Result<()> {
    cln_container.stop().await?;
    std::thread::sleep(Duration::from_secs(5));
    cln_container.start().await?;
    Ok(())
}

fn test_vls_recovery_after_node_crash() -> Result<()> {
    let docker = Docker::new()?;

    let cln_container = docker.containers().create(...)?; // CLN Config
    let vls_container = docker.containers().create(...)?; // VLS Config

    cln_container.start()?;
    vls_container.start()?;

    channel_setup(&docker, &cln_container).await?;

    // ... logic to send HTLCs ...

    inject_cln_node_crash(&docker, &cln_container).await?;

    // Assertions using potential crates or interacting with services
    // ... examine vls_container logs or directly check VLS state...

    Ok(())
}
```

- **Test Data and Assertions**
  - **Data Generation:** Design realistic yet targeted test data for channels, HTLCs, etc. Explore parameter randomization for fuzz-like testing.
  - **Data Monitoring:** Logs from all components (VLS, CLN nodes, `bitcoind`) will be aggregated into a central system for easy analysis during and after tests.
  - **Assertion Framework:**
    - Build assertion logic using Rust's standard testing constructs or a dedicated test crate.
    - Implement assertions to verify VLS state, interaction with CLN, reactions to injected failures, and the validity of justice transactions.
    - If applicable, a metrics system (potentially Prometheus-based) will be integrated to capture performance indicators under various load conditions.

Illustrative Diagram (Simplified):



- **Failure Scenario Coverage:**

  1. Node/Signer Crashes

  - **Injection Points:** Identify critical stages during the channel lifecycle, including:
    - During channel negotiation.
    - After funding transaction broadcast, but before confirmation.
    - Mid-operation with active HTLCs.
    - During the closing negotiation process.
  - **Crash Types:**
    - Graceful Shutdown: Emulate termination signals allowing VLS to save state.
    - Hard Crash: Simulate abrupt process termination.
  - **Assertions:**
    - Channel states remain consistent upon restart.
    - VLS logs indicate successful recovery or detail the issue if recovery fails.
    - Channel counterparties are notified appropriately (if applicable).

Code Snippet (Illustrative):

```rust
use docker_api::Docker;
use std::time::Duration;

#[test]
fn test_vls_recovery_crash_during_negotiation() -> Result<(), Box<dyn std::error::Error>> {
    let docker = Docker::new()?;
    let cln_container = start_cln_node(&docker)?;
    let vls_container = start_vls(&docker)?;

    // ... Initiate channel negotiation

    // Crash mid-negotiation!
    vls_container.kill()?;
    std::thread::sleep(Duration::from_secs(3)); // Allow state change
    vls_container.start()?;

    // Assertions
    assert_vls_log_indicates_negotiation_failure()?;
    assert_channel_not_established(&cln_container)?;

    Ok(())
}
```

2. Data Loss and Corruption

- **Targets:**
  - VLS's internal database or state files.
  - Blockchain data (simulating corruption within `bitcoind`).
- **Methods:**
  - File modification/deletion at critical moments.
  - Utilize tools like `fault_injection` (may require modifications for VLS data stores).
- **Assertions:**
  - Graceful failure with error logging if data is unrecoverable.
  - Detection of corrupted state with appropriate actions (rollback, alerts).

Code Snippet (Illustrative):

```rust
use std::fs;
use rand::{thread_rng, Rng};

#[test]
fn test_data_corruption_with_htlcs() -> Result<(), Box<dyn std::error::Error>> {
    // ... Setup channel, offer and accept HTLCs

    let data_file = vls_state_file_path();
    let mut data = fs::read(data_file)?;

    // Corrupt a random section of the data
    let corrupt_range = thread_rng().gen_range(0..data.len() / 2);
    data[corrupt_range].fill(0xFF);  // Overwrite with arbitrary bytes
    fs::write(data_file, data)?;

    restart_vls_container();

    // Assertions (depend on VLS-specific error handling)
    assert_vls_log_indicates_corruption()?;
    assert_htlc_states_failed_or_pending()?;
}
```

3. Power Outages

- **Varying State Loss:** Simulate power loss at different stages:
  - Incomplete writes to disk.
  - Data fully persisted but in-memory state lost.
- **Replication:** If VLS employs data replication, test scenarios where:
  - The primary instance fails.
  - Both primary and replica fail simultaneously.
- **Assertions:**
  - Upon restart, the integrity of persisted data is verified.
  - Data replication mechanisms function as designed.

Code Snippet (Illustrative):

```rust
use docker_api::Docker;
use std::thread::sleep;
use std::time::Duration;

#[test]
fn test_vls_recovery_after_power_outage() -> Result<(), Box<dyn std::error::Error>> {
    // ... Setup channel with active HTLC

    let docker = Docker::new()?;
    let vls_container = start_vls(&docker)?;

    vls_container.stop()?; // Simulate hard power loss
    sleep(Duration::from_secs(5));
    vls_container.start()?;

    // Assertions
    assert_vls_logs_indicate_clean_start()?;
    assert_htlc_states_resumed_or_resolved(&vls_container)?;

    Ok(())
}
```

4. Cooperative vs. Uncooperative Closure

- **Cooperative:**
  - Test the standard mutual close flow.
  - Varying fee preferences and negotiation.
- **Uncooperative:**
  - Induce delayed or absent close responses from counterparty.
  - Provoke the broadcast of commitment transactions.
  - Test with HTLCs in various stages (offered, accepted, resolved).
- **Assertions:**
  - Correct closing transaction structure.
  - Appropriate use of justice transactions when necessitated.

Code Snippet (Illustrative):

```rust
use tokio::time::timeout;

#[test]
async fn test_uncooperative_close_with_htlc() -> Result<(), Box<dyn std::error::Error>> {
    // ... Setup channel with pending HTLC

    trigger_channel_close(&vls_container, &cln_container)?;

    // Introduce artificial delay or simulate unresponsiveness from CLN

    let result = timeout(Duration::from_secs(30),
assert_commitment_broadcast(&vls_container)).await;

    match result {
        Ok(()) => { /* Success, commitment tx seen */ },
        Err(_) => { /* Timeout - potentially examine logs for errors */ }
    }

    Ok(())
}
```

5. Recovery Procedures

- **State Variation:** Generate channels at different stages:
  - Funded, but inactive.
  - HTLCs in flight, different timeout values.
  - Pending unilateral close.
- **Triggering Recovery:** Simulate crash/power loss during each state.
- **Assertions:**
  - VLS successfully resumes operation.
  - Outstanding HTLCs are resolved or appropriate actions taken based on their state.

Code Snippet (Illustrative):

```rust
// Note: Simplified for illustration, assumes modularity

#[test]
fn test_recovery_funded_channel() {
    let state = setup_channel_funded_inactive();
    simulate_crash_and_recover();
    assert_channel_state_consistent(state);
}

#[test]
fn test_recovery_channel_pending_close() {
    let state = setup_channel_with_pending_close();
    simulate_crash_and_recover();
    assert_close_completed_or_error(state);
}
```

- **Stress Testing:**

1. Identifying Performance Bottlenecks

- **Profiling:** Utilize profiling tools (both within Rust, and system-level if applicable) to pinpoint potential hotspots within VLS under normal load. These could involve:
    - Cryptographic operations.
    - State database interactions.
    - Network communication with CLN or other nodes.

Example (Profiling with `pprof`):

```
# Build VLS with profiling support
RUSTFLAGS="-C target-cpu=native -g" cargo build

# Run a stress test (replace with your specific test)
cargo test stress_test_channel_churn -- --nocapture

# Generate pprof report (text format in this example)
pprof --text ./target/debug/vlsd2 <stress test data file> > profile.txt
```

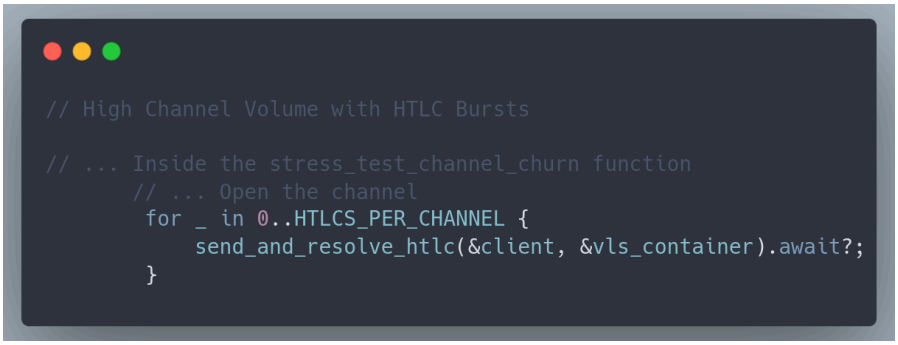
- **Baseline Metrics:** Establish baseline performance metrics under various loads:
    - Channel opening/closing latency.
    - HTLC throughput.
    - Memory usage.
    - CPU utilization.

Example (Baseline Metrics with Prometheus):

```
use prometheus::{HistogramOpts, Histogram, register_histogram};

let channel_open_latency = register_histogram!(
    HistogramOpts::new("vls_channel_open_latency", "Channel open latency (seconds)")
    .buckets(vec![0.01, 0.05, 0.1, 0.5, 1.0, 2.0]),
    "Total time to open a channel").unwrap();

// ... Inside channel opening logic
let timer = channel_open_latency.start_timer();
// ... Perform channel opening steps
timer.observe_duration();
```

2. Stress Test Scenarios

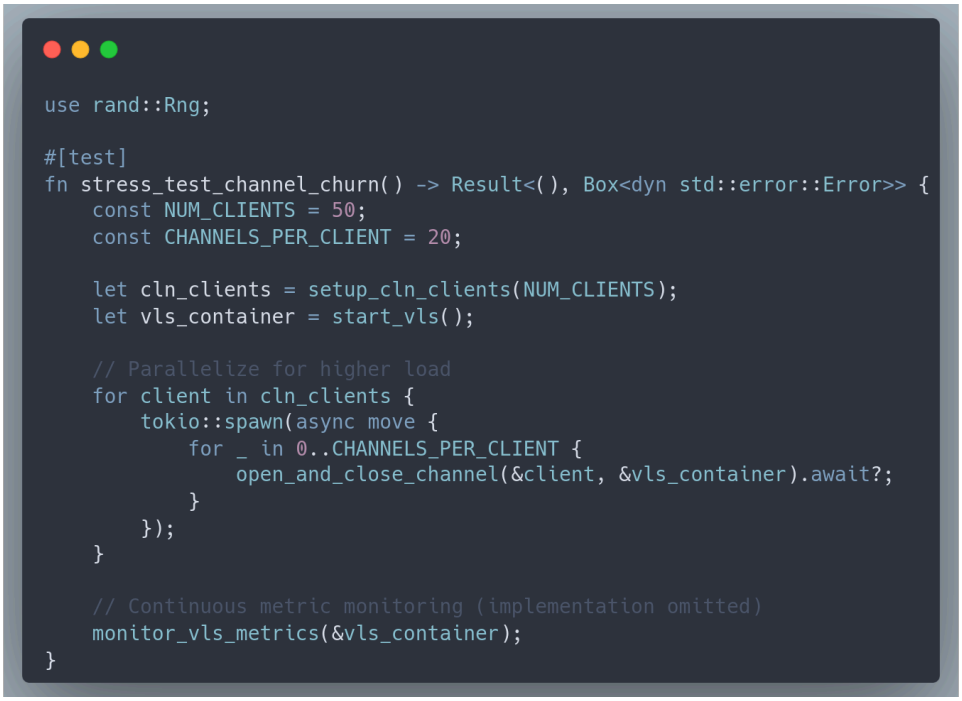
- **High Channel Volume:**
    - Simultaneously open and maintain a large number of channels (more than normal usage).
    - Push higher than normal HTLCs per channel and across many channels.



```
// High Channel Volume with HTLC Bursts

// ... Inside the stress_test_channel_churn function
       // ... Open the channel
        for _ in 0..HTLCS_PER_CHANNEL {
            send_and_resolve_htlc(&client, &vls_container).await?;
        }
```

- **Rapid Activity Bursts:**
  - Subject VLS to sudden spikes in channel opening requests or HTLC activity.
- **Resource Constraints:**
  - Introduce artificial limits on CPU, memory, or disk I/O to identify bottlenecks.
- **Edge Case Combinations:** Combine the above with failure scenarios:
  - Node crashes during periods of high activity.
  - Data corruption under load.

Code Illustration:



```
use rand::Rng;

#[test]
fn stress_test_channel_churn() -> Result<(), Box<dyn std::error::Error>> {
    const NUM_CLIENTS = 50;
    const CHANNELS_PER_CLIENT = 20;

    let cln_clients = setup_cln_clients(NUM_CLIENTS);
    let vls_container = start_vls();

    // Parallelize for higher load
    for client in cln_clients {
        tokio::spawn(async move {
            for _ in 0..CHANNELS_PER_CLIENT {
                open_and_close_channel(&client, &vls_container).await?;
            }
        });
    }

    // Continuous metric monitoring (implementation omitted)
    monitor_vls_metrics(&vls_container);
}
```

3. Test Harness

- **Scenario Definition:** Develop complex scenarios involving multiple channels, sequences of operations, and potentially parallel test execution.
- **Load Generation:**
  - Multiple CLN instances interacting with VLS.
  - Consider specialized load testing tools to introduce network delays and bottlenecks.
- **Metric Capture:** Integrate continuous metric collection during tests (using Prometheus).

Code Illustration:

```rust
use docker_api::Docker;
use rand::Rng;
use std::sync::Arc;
use tokio::{spawn, time::sleep, time::Duration};

// ... metric setup with Prometheus (as covered previously)

#[derive(Debug)]
struct StressScenario {
    num_cln_clients: usize,
    channels_per_client: usize,
    htlcs_per_channel: usize,
    // ... other parameters (delays, failure injection, etc.)
}

#[test]
fn stress_test_with_scenario(scenario: StressScenario) -> Result<(), Box<dyn std::error::Error>>
{
    let docker = Docker::new()?;
    let vls_container = start_vls_container(&docker)?;

    let clients = Arc::new(setup_cln_clients(&docker, scenario.num_cln_clients)?);

    // Parallelize client load generation
    let mut handles = vec![];
    for _ in 0..scenario.num_cln_clients {
        let clients_clone = clients.clone();
        let vls_container_clone = vls_container.clone();
        let channels_to_open = scenario.channels_per_client;
        let htlcs_per_channel = scenario.htlcs_per_channel;

        handles.push(spawn(async move {
            client_stress_task(clients_clone, vls_container_clone, channels_to_open,
htlcs_per_channel).await?;
            Ok(())
        }));
    }

    // Continuous metric monitoring (in background)
    spawn(async move {
        monitor_vls_metrics(&prometheus, &vls_container).await;
    });

    // Wait for client tasks, report aggregate results
    for handle in handles {
        handle.await??; // Propagate errors from client tasks
    }

    // ... Analyze collected metrics, logs, etc.
    Ok(())
}

// (Helper Functions)
async fn client_stress_task(
    clients: Arc<Vec<cln::Client>>,
    vls_container: docker_api::container::Container,
    channels: usize,
    htlcs_per_channel: usize,
) -> Result<(), Box<dyn std::error::Error>> {
    // ... Logic to open channels, send HTLCs, potentially with delays/failures
    Ok(())
}

// ... setup_cln_clients, monitor_vls_metrics (implementation details omitted)
```

4. Analysis and Iteration

- **Visualization:** Employ tools to plot metrics over time, highlighting periods of stress and potential degradations.
- **Log Correlation:** Synchronize VLS logs with metrics to pinpoint issues.

```
// Using a logging crate like 'tracing'
tracing::info!(target: "vls::channel_ops", "Channel opened in {}ms", timer.elapsed().as_millis());
```

- **Targeted Optimization:**
  - Use profiling results and stress test observations to guide code optimization.
  - Iteratively run tests and assess the impact of changes made to VLS.

- **Automated Test Suite:**

1. Recovery Scenario Catalog

- **Thorough Identification:** Systematically list the various scenarios from which VLS should recover:
  - Node crashes at different channel lifecycle stages.
  - Data corruption (simulating different failure modes).
  - Unplanned power outages with varying degrees of data loss.
- **Prioritization:** Rank scenarios based on severity and likelihood. Focus initial automation on the most critical recovery scenarios.

2. Test Automation Framework (Rust)

- **Framework Selection:**
  - `libtest` (built-in) is a solid starting point.
  - Can consider crates like `serial_test` for serialization when running tests on a containerized setup.
    - This ensures tests that manipulate state run sequentially to avoid interference.
- **Assertions:**
  - VLS logs indicate successful recovery or clear error messages in case of failure.
  - Channel states are consistent.
  - Interaction with counterparty nodes (if applicable) resumes correctly.

Code Illustration:

```rust
use cln::Client; // Assuming control of CLN container

// ... after VLS restart

// VLS log analysis (library choice can enhance this)
assert!(vls_log_contains("Recovery initiated", "error.log"));

let cln_client = Client::new(&cln_container)?;
let channel_info = cln_client.listchannels(..)?;

// Assuming HTLC resolution is the success criterion:
assert!(channel_info.channels.iter()
        .find(|c| c.id == test_channel_id)
        .map(|c| c.htlcs.is_empty()) // No pending HTLCs
        .unwrap_or(false), "HTLCs not resolved");
```

3. State Manipulation and Failure Injection

- **Controlled Chaos:**
  - File modification tools to simulate data corruption precisely.
  - Container management for crashes and restarts (Docker API).
  - Network manipulation (`tc`, `iptables`) for delayed or lost responses.

Code Illustration:

```rust
use std::fs;
use rand::{thread_rng, Rng};

// ... inside a recovery test

// Targeted data corruption
let vls_data_file = vls_state_file_path();
let mut data = fs::read(vls_data_file)?;

let corrupt_offset = thread_rng().gen_range(0..data.len() / 2);
data[corrupt_offset .. corrupt_offset + 32].fill(0); // Overwrite a section
fs::write(vls_data_file, data)?;

// Crash and restart ...
```

- **State Snapshots (Optional):** For complex tests, consider snapshotting VLS state at strategic points for quicker test setup and recovery to a known good state.

Example Code (Conceptual):

```
# Snapshot format is highly dependent on VLS's internal state representation
fn snapshot_vls_state(vls_container: &docker_api::container::Container) -> Result<Vec<u8>, ...> {
    // ... Logic to serialize VLS state, potentially compress
}

fn restore_vls_state(vls_container: &docker_api::container::Container, snapshot: &[u8]) ->
Result<(), ...> {
    // ... Deserialize and populate VLS's internal data structures
}

// ... In a test:
let snapshot = snapshot_vls_state(&vls_container)?;
// ... Induce failure
restore_vls_state(&vls_container, &snapshot)?;
// ... Assertions on recovered state
```

4. Test Execution and Reporting

- **Integration:** Incorporate the recovery test suite into CI/CD pipeline.
- **Reporting:** Generate clear reports on successes/failures, ideally with links to relevant logs.

Code Illustration:

```
use docker_api::Docker;
use serial_test::serial;
use std::fs;
use std::time::Duration;

#[test]
#[serial] // Ensure this test has exclusive control
fn test_recovery_from_crash_during_active_htlc() -> Result<(), Box<dyn std::error::Error>> {
    let docker = Docker::new()?;
    let vls_container = start_vls_container(&docker)?;
    let cln_container = start_cln_container(&docker)?;

    // ... Setup channel with active HTLC (external functions assumed)

    // Simulate Crash!
    vls_container.kill()?;
    sleep(Duration::from_secs(3)); // Allow state change
    vls_container.start()?;

    // Assertions
    assert_vls_log_indicates_recovery(vls_container)?;
    assert_htlc_resolved_or_failed(cln_container, vls_container)?; // Implementation depends on
specifics

    Ok(())
}
```

3. Documentation and Guidelines:

- **Categorizing Recoverable Scenarios**

- **Severity-Based:** Divide scenarios into major categories (e.g., simple restart, potential data loss, hardware failure). This helps establish the urgency and complexity level for users.
- **Failure Types:** Group by failure triggers:
    - Node crashes/unavailability
    - Data corruption (disk, file-specific, etc.)
    - External dependency issues (e.g., `bitcoind` unresponsiveness)

- **Multi-Level Guidance**
    - **Quick Start for Basic Users:**
        - Concise, step-by-step instructions focused on the most common and easily resolved scenarios.
        - Emphasize identifying the issue (e.g., checking logs for specific keywords).
        - Include when to escalate to more advanced support.
    - **Technical Troubleshooting Guides:**
        - In-depth explanations of VLS recovery mechanisms.
        - Troubleshooting flowcharts to aid in diagnosis.
        - Potential commands or tools for data inspection and repair (if applicable).
    - **Developer Reference (Optional):**
        - Internal mechanisms, data structure considerations, and code references for those extending or debugging VLS itself.

- **Documentation Structure**
    - **Centralized Hub:** A dedicated section within the VLS project's documentation.
    - **Sections:**
        - Backup Procedures
        - Restoration Procedures
        - HTLC Sweeps
        - Scenario-Specific Recovery
        - Troubleshooting
    - **Clarity:** Use plain language whenever possible, with technical terms explained in a glossary.
    - **Examples:** Include concrete examples for common errors and recovery commands.
    - **Visual Aids (Optional):** Consider diagrams to illustrate component relationships or recovery sequences.

- **Format and Integration**
    - **Primary Format:** Markdown for easy integration into VLS documentation system.
    - **Cross-Referencing:** Link recovery documentation to relevant sections within the main VLS documentation.
    - **Automation:** Where possible, provide pre-made scripts or tools to streamline recovery steps.

- **Detailed Sections**

I. Backup Procedures

- **What to Back Up:**
    - VLS database or state files.
    - VLS configuration (`vlsd2.toml`).
    - Relevant `bitcoind` data (wallet, blockchain data depending on setup).
- **Frequency:** Recommendations based on usage patterns and risk tolerance.
- **Methods:**
    - Manual or automated scripts.
    - Integration with cloud backup solutions.
- **Security:**
    - Encryption for sensitive data.
    - Secure storage of backups (off-site replication if necessary).

II. Restoration Procedures

- **Step-by-Step Instructions:**
    - Prerequisites (VLS version compatibility, etc.)
    - Restoring VLS data/configuration.
    - Re-establishing dependencies (CLN, `bitcoind`).
- **Data Consistency Checks:**
    - Verification tools or processes to ensure backup integrity before restoration.

III. HTLC Sweeps

- **Conditions:** Clearly define when HTLC sweeps are necessary.
    - VLS unreachable for an extended period.
    - Data corruption specifically affecting HTLC state.
- **Tools/Mechanisms:**
    - Does VLS provide built-in functionality?
    - If manual interaction with `bitcoind` is required, document the commands/process.
- **Security Emphasis:**
    - The need for heightened caution when constructing sweep transactions.

IV. Scenario-Specific Recovery

- **Prioritization:** Map out recovery guides for the most likely or severe scenarios:
    - Crash during channel opening.
    - Data loss due to disk failure.
    - Unplanned power outages.
- **Decision Trees (if applicable):** Flowcharts can aid in troubleshooting and guide the choice of recovery actions.

V. Troubleshooting

- **Log Analysis:** Guidance on where to find logs (VLS, CLN, `bitcoind`) and what errors to look for.

- **Common Issues:** Document known problems and solutions.
- **Support Channels:** Where to seek help if needed.

- **Best Practices**

  Some possible best practices to include:

  1. **Security Guidelines**
     - **HSM Integration**: Utilize HSM with VLS for key protection and secure operations.
     - **Channel Counterparty Selection**: Choose reputable nodes with sufficient financial capacity.
     - **Threat Modeling**: Be aware of potential threats like compromised nodes and network attacks.
  2. **Redundancy and Fault Tolerance**
     - **Multiple VLS Instances**: Set up redundant VLS instances with load balancing or active/standby configurations.
     - **bitcoind Redundancy**: Maintain multiple bitcoind nodes for availability.
     - **Data Replication**: Regularly back up VLS's database or state files, both locally and off-site.
  3. **Configuration Best Practices**
     - **Chain Backend**: Choose bitcoind or other backends based on security/resource tradeoffs.
     - **Lightning Network Parameters**: Select appropriate values for channel timeouts, fee policies, etc.
     - **Network Monitoring**: Use tools to detect unresponsive nodes or unusual network conditions.

4. Fault Tolerance Strategies:

I. Redundancy Options

- **Full Replication:**

  - **Pros:** Fastest failover, protects against complete data loss.
  - **Cons:** Higher resource cost, potential for state synchronization lag.
  - **Implementation:**
    - Real-time replication of VLS's state database.
    - Load balancing or health checks for failover orchestration.

Example (Conceptual):

```rust
use std::sync::Arc;
use tokio::sync::Mutex;

// ... (error handling simplified for brevity)

#[derive(Clone)]
struct VLSState(Arc<Mutex<InternalVLSState>>);

async fn replicate_state(secondary_vls: &VLSState, state: VLSState) {
    let mut secondary_state = secondary_vls.0.lock().await;
    *secondary_state = state.0.lock().await.clone();
}

// Failover Logic (Simplified)
async fn failover_to_secondary(secondary_vls: &VLSState) {
    let mut secondary_state = secondary_vls.0.lock().await;
    // ... Logic to reconfigure secondary_state for primary role
}

// ... inside run_vls_with_replication
if let Err(e) = primary_vls_healthcheck().await {
    failover_to_secondary(&secondary_vls).await;
}
```

- **Partial Replication:**
    - **Pros:** Reduced overhead, focuses on essential recovery data.
    - **Cons:** Slower failover as some state may need to be rebuilt.
    - **Implementation:**
        - Identify core channel state required for recovery.
        - Tailored replication mechanism, likely more fine-grained than full state.

Example (Conceptual):

```rust
#[derive(Serialize, Deserialize)] // Assume serde for serialization
struct CriticalChannelData {
    channel_id: ChannelId,
    recent_commitments: Vec<Commitment>,
    // ... other essential elements
}

async fn replicate_critical_state(secondary: &SecondaryStorage, data: CriticalChannelData) {
    secondary.store_channel_data(data).await;
}
```

- **Checksums:**
    - **Pros:** Lightweight, early corruption detection.
    - **Cons:** Doesn't prevent corruption, merely alerts to it.

- **Implementation**
  - Utilize a strong cryptographic hash function (e.g., SHA-256).
  - Periodically calculate checksums of VLS's state file(s).
  - Store checksums separately (ideally in a redundant fashion).

Example with Alerting (Conceptual):

```rust
// ... (verify_vls_state_integrity as before)

fn monitor_vls_integrity() -> Result<(), Box<dyn std::error::Error>> {
    loop {
        verify_vls_state_integrity()?;
        tokio::time::sleep(Duration::from_secs(300)).await; // Check interval
    }
    Ok(())
}

// In startup logic:
tokio::spawn(async move {
    if let Err(e) = monitor_vls_integrity() {
        error!("VLS integrity compromised, alerting operator: {}", e);
        // Trigger alerts, administrator intervention needed
    }
});
```

II. Graceful Shutdown

- **Integration with Lightning Node**
  - **RPC Communication:**
    - Utilize CLN's RPC interface (or equivalent) to subscribe to shutdown notifications.
    - Implement a handler within VLS to receive these notifications.
  - **Error Handling:** Consider scenarios where the node might shut down unexpectedly.

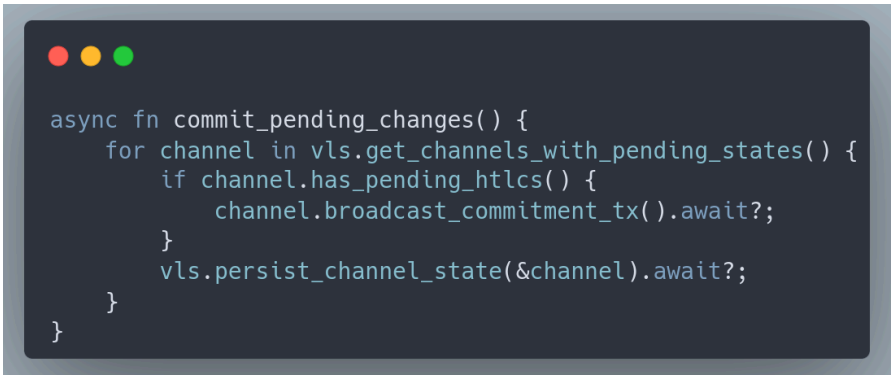
Snippet (Illustrative)

```rust
use cln::Client; // Assuming control of CLN

// During VLS initialization
let cln_client = Client::new(&cln_container)?;
cln_client.subscribe_shutdown(); // Assuming CLN plugin implementation

// ... (VLS setup)

async fn handle_node_shutdown_notification() {
    initiate_graceful_vls_shutdown().await;
}
```

- **Committing Pending Changes**
  - **Identifying In-Flight States:**

        - Track which channels have pending HTLCs (offered, accepted).
        - Determine if commitments have been exchanged but not fully confirmed.
    - **Commit Mechanisms:**
        - Logic to force the broadcast of pending commitment transactions, where applicable.
        - Ensure VLS persists the resulting state changes before proceeding.

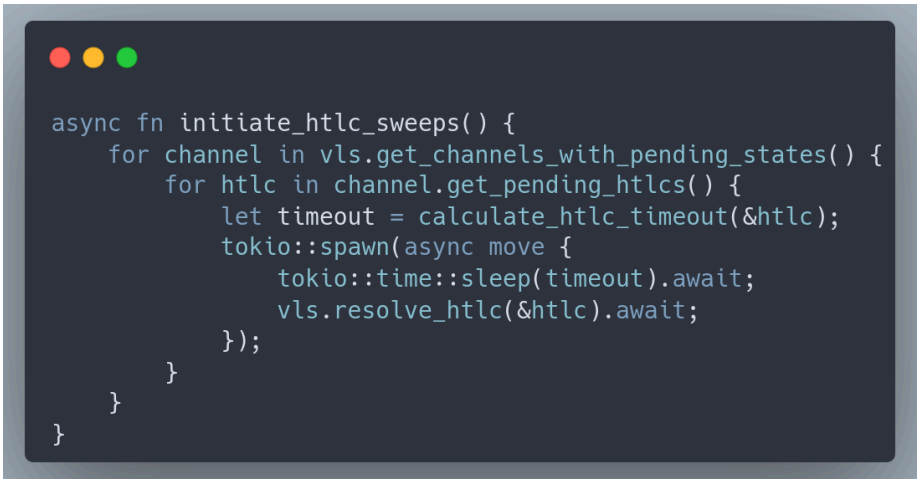
Snippet (Conceptual)



```
async fn commit_pending_changes() {
    for channel in vls.get_channels_with_pending_states() {
        if channel.has_pending_htlcs() {
            channel.broadcast_commitment_tx().await?;
        }
        vls.persist_channel_state(&channel).await?;
    }
}
```

- **Initiating HTLC Sweeps**
    - **Sweep Timeouts:**
        - Calculate appropriate timeouts based on CLTV or CSV values in outstanding HTLCs.
        - Provide some margin to account for potential propagation delays.
    - **Sweep Initiation:**
        - Trigger the HTLC resolution process within VLS, potentially using existing sweep logic.

Snippet (Conceptual)



```
async fn initiate_htlc_sweeps() {
    for channel in vls.get_channels_with_pending_states() {
        for htlc in channel.get_pending_htlcs() {
            let timeout = calculate_htlc_timeout(&htlc);
            tokio::spawn(async move {
                tokio::time::sleep(timeout).await;
                vls.resolve_htlc(&htlc).await;
            });
        }
    }
}
```

Graceful Shutdown Orchestration (Conceptual)

```
async fn initiate_graceful_vls_shutdown() {
    commit_pending_changes().await;
    initiate_htlc_sweeps().await;
    // ... Final cleanup, logging
}
```

- **Enhanced Logging:**
  - **Transaction Timeouts**
    - **CLTV (CheckLockTimeVerify):** Dictates the earliest block height at which a transaction is valid.
    - **CSV (CheckSequenceVerify):** Imposes conditions based on the relative number of blocks elapsed since a previous output was recorded.
    - **HTLC Timeouts:** Predetermined values included in HTLC outputs, setting time constraints on resolution.
  - **Tracking Timeouts**
    - **State Storage Augmentation:**
      - Persist CLTV or CSV values alongside pending HTLCs in VLS's channel state representation.
      - Store absolute block expiry (if using CLTV) for straightforward calculations during sweeps.

Snippet (Conceptual)

```
#[derive(Serialize, Deserialize)]
struct PendingHtlc {
    // ... existing fields
    cltv_expiry: Option<u32>, // Absolute block height
    csv_expiry: Option<u32>, // Relative block number
}

fn calculate_time_remaining(htlc: &PendingHtlc, current_block_height: u32) -> Duration {
    let expiry = match (htlc.cltv_expiry, htlc.csv_expiry) {
        (Some(cltv), _) => cltv,
        (_, Some(csv)) => current_block_height + csv,
        _ => panic!("HTLC missing expiry information"), // Robustness
    };

    let blocks_remaining = expiry.saturating_sub(current_block_height);
    let time_remaining = Duration::from_secs(blocks_remaining * AVG_BLOCK_TIME);
    time_remaining - SAFETY_MARGIN
}
```

  - **Timeout-Aware Sweep Logic**
    - **Dynamic Timeout Calculation:**
      - During sweep attempt, calculate remaining time until HTLC expiry.
      - Set the CLTV or CSV in the sweep transaction to allow sufficient propagation before the HTLC's timeout.

- **Safety Margin:** Include a configurable buffer to account for potential blockchain congestion or delays.

Snippet (Conceptual)

```rust
async fn attempt_sweep_htlc(htlc: &PendingHtlc) -> Result<(), ...> {
    let current_block_height = get_current_block_height()?;
    let time_remaining = calculate_time_remaining(htlc, current_block_height);

    let sweep_tx = build_sweep_transaction(&htlc);

    // Set appropriate CLTV or CSV based on htlc.cltv_expiry or htlc.csv_expiry
    set_timeout_on_transaction(&sweep_tx, time_remaining);

    broadcast_transaction(&sweep_tx).await?;
    Ok(())
}
```

- **Justice Transactions with Timeouts**
  - **Revocation Secret Knowledge:**
    - Assuming a mechanism within VLS to retrieve revocation secrets for expired HTLCs it offered.
  - **Timeout Calculation:** Similar to sweep logic, factor in the HTLC's timeout when creating justice transactions.

5. Error Analysis and Logging

- Expand the VLS logging system to include:
  - **Precise timestamps**
    - **High-Resolution:**
      - Utilize a timestamp format with millisecond or better precision for accurate error sequencing.
    - **Synchronization (if applicable):**
      - When VLS interacts with other components (CLN, bitcoind), ensure consistent time sources for meaningful correlation.

```rust
use chrono::{Local, DateTime, Utc};

fn get_current_timestamp() -> String {
    let now: DateTime<Local> = Local::now();
    now.format("%Y-%m-%dT%H:%M:%S%.3f").to_string() // Example format
}

// ... inside logging macros
error!(timestamp = get_current_timestamp(), error_code = ?ErrorCode::HTLCResolutionTimeout,
"...");
```

- **Granular error codes - Example**

```rust
#[derive(Debug, Clone)]
pub enum VLSChannelError {
    InvalidStateTransition,
    CommitmentBroadcastFailure(String),
    HTLCResolutionTimeout(HtlcId),
    // ... Add more specific errors within the channel context
}

impl std::error::Error for VLSChannelError { }

impl std::fmt::Display for VLSChannelError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            VLSChannelError::InvalidStateTransition => write!(f, "Invalid channel state
transition"),
            VLSChannelError::CommitmentBroadcastFailure(msg) => write!(f, "Commitment transaction
broadcast failed: {}", msg),
            VLSChannelError::HTLCResolutionTimeout(htlc_id) => write!(f, "HTLC resolution timed
out (HTLC ID: {})", htlc_id),
            // ... Format other errors
        }
    }
}

// ... Similar structure for VLSError
```

- **Transaction and Channel State Context**
  - **Relevant Identifiers:** Include the following in error logs (where applicable):
    - Channel IDs
    - Transaction IDs (HTLCs, commitments, etc.)
    - Specific HTLC identifiers
  - **State Snapshots (Selective):**
    - For complex errors, log a serialized representation of the relevant channel state. Consider anonymizing addresses or private keys if necessary.

```rust
use serde_json; // Or another serialization library

// ... when a complex error condition is encountered
if should_include_state_snapshot() { // Criteria based on error type, etc.
    let channel_state = vls.get_channel_state(&channel_id);
    let sanitized_state = anonymize_sensitive_data(&channel_state);
    error!(?sanitized_state, "Channel state at time of error");
}
```

Timeline

I aim to make weekly to biweekly MRs for the project so that there is enough time for the mentors to review all the changes and suggest improvements before the next sprint cycle. This would also ensure we incrementally

develop the component continuously and do not block progress due to unstable requirements and changes for unpredictable reasons.

Period	Tasks
After proposal submission [May 1 - May 14]	- Research additional requirements for the project - Convert the requirements to a technical requirement document for easier development in the later phases - Research other libraries and projects about how they address same concerns - Begin with the initial implementation of the components

Weeks 1-2 [May 15 - May 28]	- **State Identification and Backup:**
	  - Thoroughly analyze VLS data structures to pinpoint the essential state.
	  - Design backup formats and serialization.
	  - Implement secure backup logic (encryption, key management).
	- **HTLC Handling (Foundation):**
	  - Introduce in-memory HTLC tracking.
	  - Augment state persistence to include HTLCs.

Weeks 3-4 [May 29 - June 11]	- **Restoration Procedures:**
	  - Develop core state restoration functionality.
	  - Outline recovery guides.
	- **HTLC Sweeps (Outgoing):**
	  - Implement sweep logic for HTLCs where VLS has the preimage.
	  - Design basic test scenarios for failure and sweep execution.
	- **Justice Transactions (Conceptual):**
	  - Investigate revocation secret management strategies.

Weeks 5-6 [June 12 - June 25]	- **HTLC Sweeps (Incoming):**
	  - Logic for sweep attempts where VLS receives funds.
	  - Enhance test cases to cover both incoming and outgoing HTLCs.
	- **Justice Transactions (Prototyping):**
	  - Prototype justice transaction construction based on state comparison.
	- **Test Environment:**
	  - Begin building the containerized test environment.

Weeks 7-8
[June 26 - July 17]
	- **Stress Testing:**
	  - Develop stress test scenarios and initial infrastructure.
	- **Recovery Refinement:**
	  - Iterate on recovery mechanisms based on testing.
	  - Begin drafting detailed recovery procedures.
	- **Documentation (Start):**
	  - Best practices outline and initial guides.

Weeks 9-10
[July 18 - August 1]
	- **Redundancy (Exploration):**
	  - Research and prototype partial replications.
	  - Evaluate full replication if resources permit.
	- **Graceful Shutdown (Core):**
	  - Implement RPC hooks and commit logic.
	- **Error Analysis:**
	  - Enhanced logging with timestamps and error codes.

Weeks 11-12
[August 2 - August 15]
	- **Fault Tolerance (Continued):**
	  - Checksum implementation.
	  - Graceful shutdown with sweep initiation.
	- **Transaction Safety:**
	  - Timeout awareness in sweeps and justice transactions.
	- **Testing and Documentation Polish:**
	  - Automated tests, comprehensive guides.

Celebrate 🎉	

About Me

I am a **systems developer** with experience with multiple tech stacks including **back end, embedded systems, distributed systems** and a particular interest in **blockchain and lightning networks**. I have been fortunate enough to contribute to numerous OpenSource organizations, which have helped me learn so much and diversify my tech stack!

I am a responsible individual and a great team player who believes in the utmost importance of active communication. I consider myself fortunate to have stumbled upon Validating Lightning Signer as a project and as a community as my last couple months with VLS has been nothing but a bliss. Such helpful mentors who took a meeting just to greet me and help me set up the project were so pleasantly new to me. Also, being helped and checked upon ever so regularly felt so motivating and made me want to learn more and contribute as much as I can. VLS has been an experience that makes me want to become a better developer and a person in general which I am so grateful for.

Rust has quickly become my go-to language for its blend of power, safety, and efficiency. I'm constantly inspired by the vibrant open-source community around Rust, and I believe contributing and learning from others is the best way to grow as a developer. In the past few months, I've actively participated in several **Rust-based projects**. **VLS** is a project I feel the most attracted towards as stated above and hence I want to contribute to the project as much as I can going forward.

Here are the links to my contributions to various organizations written in **Rust** in the **last ~2 months**:

- **Validating Lightning Signer**: PR #646, PR #636, PR #633, PR #632, PR #627

- **Xline**: PR #682, PR #676, Issue #672
- **Cord**: PR #352, PR #351, PR #346, PR #343, PR #342, Issue #323
- **Icu4x:** PR #4628

Commitments

I have no other commitments for the upcoming summer and the Summer Of Bitcoin timeline. I will not have any tests or examinations between this period, or even after 30 days of completion, hence I can give my undivided attention to this project and nothing else.

Plans after Summer Of Bitcoin

I love OpenSource, and working with the Validating Lightning Signer team has been one of my favorite experiences. The fact that the team took the time to actively discuss and deliberate on even the simplest of my doubts along the way is special to me! I would love to continue to be an active community member here as a full-time code contributor and a guide who can help other Open Source enthusiasts stumbling upon the VLS ecosystem. I also plan on refining this project furthermore by working on the stretch goals mentioned earlier. I hope to grow together with VLS as it has been the most welcoming organization I've worked with and I look forward to sticking with it for years to come.

Planning for the Development Period

I would like to work with the mentors to decide weekly goals to align myself and work in short sprints to build the required functionality. The timeline for the project, as mentioned earlier, would help me guide the overall development.

I am a good communicator, so I want to stay in close sync with the mentors via texts, Matrix, or meetings to clear all the blocking factors quickly and focus more on development. I am a night owl, and I plan to devote 6-8 hours every day, six days a week, to the project for 12 weeks, totaling more than 350 hours of work. I would happily put in extra hours if the project demands the same.

Appendix

Competency Test

With RUST_BACKTRACE=1

https://gist.github.com/Harsh1s/cb4f75011e2d861ad7527a066006d462

With RUST_BACKTRACE=full

https://gist.github.com/Harsh1s/a244350142ac7dbd9c8a67990229d165

Summer of Bitcoin Assignment

I completed my SoB Assignment in Rust and obtained a score of 101. (Repo made private by SoB itself)

https://github.com/SummerOfBitcoin/code-challenge-2024-Harsh1s