Audit report

Gear 2025 Q1: Gear Bridges - Bridge Components

Authors:

Andrija Mitrović

Kosa Nenadić

Marko Jurić

Miroslav Stefanović

Mile Kordić

November, 2024

Contents

Project Overview

Scope

In January 2025, Gear engaged Ethernal to continue the partnership and conduct a security audit of the gear bridge components, more specifically:

- *gear-bridges/ethereum/src (excluding ethereum/src/libraries)*
- *gear-bridges/gear-programs (excluding proof-storage and vft-service)*

The audit focused on evaluating the correctness and security properties of the implementation of the given gear bridge components.

The audit was performed from December 26th, 2024 to January 31th, 2025 by the following personnel:

- Andrija Mitrovic
- Kosa Nenadic
- Marko Juric
- Miroslav Stefanovic
- Mile Kordic

Relevant Code Commits

The audited code was from: 8b1018bd

Conclusion

The Ethernal team finds the Gear bridge's codebase to be generally well-structured, with a modular architecture that organizes bridging functionalities into multiple programs. This is done due to the fact that bridge needs to be designed as a system where a business transaction could span over multiple applications. In other words, a single business transaction may involve several local transactions within different applications. This challenge in the Gear's bridge implementation is addressed using a pattern called Saga. This microservices-like approach improves separation of concerns but introduces additional complexity in maintaining atomicity across transactions. Since Gear protocol does not guarantee atomicity by default, some state changes may require to be manually reverted if a process is not fully completed, increasing operational risks.

The audit faced challenges due to the complexity of the Gear protocol, particularly regarding its low-level APIs, asynchronous execution, error handling, and transaction finality. All that facts significantly increased the difficulty of verification and validation, as it necessitated frequent discussions with the development team.

Enhancing the protocol's documentation with more in-depth technical explanations and concrete examples (how-to, checklists, etc) would facilitate more efficient audits and improve the security posture of the ecosystem.

To ensure a more comprehensive assessment of potential vulnerabilities, an additional review of the relayer component in conjunction with the checkpoint-light-client (especially when the light client is refactored to use sails) is recommended as well as an review of the underlining Gear Protocol. This would give better insights into the bridge infrastructure and further Vara development, and help to uncover any overlooked vulnerabilities.

In summary, this audit identified a total of 7 issues, including one Critical and one High-severity vulnerability. The Gear team successfully addressed four of the reported issues, including the most critical ones. Following a thorough review, the audit team confirmed that the implemented fixes are satisfactory.

Audit Dashboard

Target Summary

- Name: Gear 2025 Q1: Bridge Components
- Code and specification version: 8b1018bd8a2b882a2b9d4fc84c12666144b54efd
- Platform: Rust, Solidity

Engagement Summary

- Dates: December 26th, 2024 to January 31th, 2025
- Method: Manual code review, protocol analysis

Findings Severity Summary

Finding Severity	#
Critical	1
High	1
Medium	1
Low	2
Informational	2
Total	7

Findings Category Breakdown

Finding Category	#
Protocol	2
Implementation	5
Code structure	-
Documentation	-
Total	7

System overview

The **Gear Bridge** is a trustless, zero-knowledge (ZK)-based cross-chain bridge designed to facilitate seamless asset transfers between Ethereum and Gear-based blockchains, such as the Vara network. The **Gear ↔ Ethereum** transfer protocol enables relaying of messages containing token transfer data between these networks. While the protocol does not guarantee the order in which messages are relayed, its robust architecture ensures efficient and secure cross-chain communication.

The bridge consists of components deployed on both chains—programs on Vara and smart contracts on Ethereum—as well as off-chain components, including relayers. These relayers are responsible for transmitting transactions involving token transfers between chains. At least one relayer incorporates a ZK prover component, which generates cryptographic proofs of events that occur on the Vara blockchain. Together, these components ensure reliable and secure bridging functionality.

This section provides an overview of the on-chain components involved in the bridging process and subject to this audit. It also explains the interactions between these components when sending or receiving transactions across the two chains. The description begins with the components and workflows on the Vara blockchain, followed by those on Ethereum.

Vara Components and Flow

Vara to Ethereum Flow

The process of bridging tokens from **Vara** to **Ethereum** involves several key components, including the **VFT Manager**, **Bridging Payment**, **Extended-VFT,** and **Wrapped-Vara** programs. These components work together to facilitate fee management, token transfer, and token compatibility with the Ethereum network.

Bridging Payment Program: Fee Management

The **Bridging Payment** program is responsible for managing the collection of fees required to incentivize relayers for performing the cross-chain operation. When a user initiates a bridging request, the **Bridging Payment** program collects the fee and forwards the request to the **VFT Manager** to handle the token transfer.

It is important to note that utilizing the **Bridging Payment** program is optional. Users can bypass it and directly interact with the **VFT Manager** to initiate a token transfer. In this case, the user will manually relay the request to Ethereum, bypassing the automatic fee collection and relayer management by the **Bridging Payment** program.

VFT Manager: Token Burning/Locking and Request Bridging

The **VFT Manager** plays a central role in the transfer of tokens from **Vara** to **Ethereum**. Once the bridging fee is handled (either via the **Bridging Payment** program or directly via the user), the **VFT Manager** handles the burning or locking of the corresponding tokens on the **Vara** network.

Before the request reaches the **VFT Manager**, it is expected that the user has granted the necessary allowances to the **VFT Manager** for either the **Extended-VFT** or **Wrapped-Vara** program (if used). These allowances are required for the **VFT Manager** to burn or lock the tokens, depending on the user's choice.

The **VFT Manager** uses the `request_bridging` method to manage the token transfer, where it processes the following transaction details:

- **Sender**: The address initiating the transfer.
- **Vara Token ID**: The type of token being transferred.
- **Amount**: The number of tokens to be transferred.
- **Recipient**: The recipient's address on **Ethereum**.

Once the tokens are burned or locked on **Vara**, the **VFT Manager** forwards the bridging request to the **pallet-gear-eth-bridge** built-in actor. This actor is responsible for processing the message further.

Extended-VFT Program: ERC20 Token Compatibility

When the **Extended-VFT** program is used, it is responsible for managing the wrapping of the **Vara** tokens to comply with the **ERC20** standard on **Ethereum**. While the **VFT Manager** handles the burning or locking of the tokens on **Vara**, the **Extended-VFT** program interacts with the Ethereum-side system to ensure that the wrapped tokens minted on **Ethereum** follow the ERC20 standard. This ensures that they are fully compatible with Ethereum-based decentralized applications and wallets that require ERC20-compliant tokens.

Wrapped-Vara Program: Native Token Wrapping

The **Wrapped-Vara** program handles native tokens on **Vara**. When a user wants to bridge native tokens, they first lock the native tokens within the **Wrapped-Vara** program. This creates wrapped versions of the native tokens on **Vara**, which can then be bridged to **Ethereum**.

When bringing the wrapped tokens back from **Ethereum** to **Vara**, the **Wrapped-Vara** program ensures that the native tokens are unlocked on **Vara** after the wrapped tokens are burned on **Ethereum**. This program ensures the interoperability of native **Vara** tokens by first wrapping them for transfer to **Ethereum** and later unwrapping them when returned.

Path Vara to Ethereum Integration

In this process, the interaction between the **Vara** network and **Ethereum** occurs as follows:

1. **User Grants Allowances**: Before initiating a bridging request, a user must grant the necessary allowances to the **VFT Manager** for either the **Extended-VFT** or **Wrapped-Vara** program. These allowances enable the **VFT Manager** to burn or lock the tokens on **Vara** as needed.
2. **User Initiates Bridging Request**: In a separate transaction, the user starts the bridging process by initiating a bridging request. This request could either involve the **Bridging Payment** program for fee management or directly interact with the **VFT Manager**.
3. **Bridging Payment Program (Optional)**: If the **Bridging Payment** program is used, it collects the required fee from the user and forwards the request to the **VFT Manager** for processing. This ensures the fee is collected to incentivize relayers before the **VFT Manager** handles the token transfer.
4. **VFT Manager Handles Token Burning/Locking**: In a separate transaction, once the request reaches the **VFT Manager**, the manager burns or locks the tokens on **Vara** calling **Extended-VFT** or **Wrapped-Vara**, depending on the direction of the transfer. This ensures that the tokens are no longer available on **Vara** and are ready to be bridged to Ethereum.
5. **Forward Request to Pallet-Gear-Eth-Bridge**: After the tokens are locked or burned, the **VFT Manager** forwards the bridging request to the **pallet-gear-eth-bridge** built-in actor.

Ethereum to Vara Flow

The **Checkpoint Light Client**, **Ethereum Event Client**, and **Historical Proxy** are interconnected programs on **Vara/Gear**, designed to facilitate secure and efficient interaction with Ethereum data. Each component plays a specific role in enabling a seamless bridge between Ethereum and Vara/Gear.

Checkpoint Light Client

The **Checkpoint Light Client** acts as a simplified Ethereum light client. It is responsible for maintaining the **sync committee validator list** and **verifying block headers**. This client ensures that Ethereum data used by other programs within the system is both accurate and trustworthy. When programs or users need to query Ethereum data, they can request checkpoint data, such as block hashes, corresponding to specific Ethereum slots. This data is crucial for validating Ethereum transactions and forms the foundation for further processing.

Ethereum Event Client

The **Ethereum Event Client** processes Ethereum transactions by leveraging the checkpoint data from the **Checkpoint Light Client**. It verifies proofs against the checkpoint data, decodes Ethereum transaction receipts, and confirms whether they are valid. This client serves as the critical interface for processing Ethereum transactions within the system, providing users and programs with information about the success or failure of the transaction. Its role is vital in enabling secure interaction with Ethereum transactions while ensuring that proofs are validated before taking further action.

Historical Proxy

The **Historical Proxy** is responsible for managing multiple **Ethereum Event Clients**, each of which is responsible for handling a specific range of Ethereum slots. When a user or program submits a request, the **Historical Proxy** determines which **Ethereum Event Client** is responsible for the specified Ethereum slot. It then forwards the request to that client for further processing. Once the **Ethereum Event Client** processes the request, the **Historical Proxy** redirects the response back to the relevant program or user. This structure helps simplify user interactions by abstracting the complexity of selecting the correct Ethereum Event Client and ensures a smooth experience in interacting with Ethereum data.

Flow of the Ethereum to Vara Integration

- **Protocol Relayer Sends Ethereum Block Hash to the Checkpoint Light Client:**
  The protocol relayer first relays the correct Ethereum block hash to the **Checkpoint Light Client**. This ensures that the Ethereum block data being used is valid and accurate.
- **Token Relayer Emits Event to the Historical Proxy:**
  Upon receiving the Ethereum block hash, the **Token Relayer** emits an event to the **Historical Proxy**. This event triggers the next steps in the bridging process.
- **Historical Proxy Selects the Correct Ethereum Event Client:**
  The **Historical Proxy** is responsible for selecting the correct **Ethereum Event Client** based on the specific Ethereum slot corresponding to the transaction. This selection ensures that the appropriate client processes the relevant Ethereum transaction data.
- **Ethereum Event Client Processes Transaction Proofs:**
  Once the correct **Ethereum Event Client** is chosen, it processes the Ethereum transaction, verifying the transaction proofs against the checkpoint data provided by the **Checkpoint Light Client**. This step ensures that the transaction is legitimate and adheres to the verified Ethereum data.
- **Transaction Receipt Validation:**
  After the Ethereum transaction has been successfully validated, the **Ethereum Event Client** decodes the transaction receipt and confirms its success or failure. The result of this validation is then returned to the **Historical Proxy**.
- **Historical Proxy Forwards Response to VFT Manager:**
  The **Historical Proxy**, acting as an intermediary, forwards the transaction result (success or failure) to the **VFT Manager**. This ensures that the correct information is passed along for the next phase of the process.
- **VFT Manager Handles Token Minting/Unlocking:**
  Upon receiving the validated transaction proof from the **Historical Proxy**, the **VFT Manager** is triggered to perform the necessary action. Depending on the direction of the transfer, the **VFT Manager** will either mint new tokens or unlock existing tokens on the **Vara** network.

Ethereum Components and Flow

Vara to Ethereum Flow

The process of bridging tokens from **Vara** to **Ethereum** involves several key components, including the **Relayer**, **Verifier**, **MessageQueue**, and **ERC20Manager** smart contracts. These components work together to facilitate token transfer.

Relayer smart contract: storing block numbers and corresponding Merkle roots

The **Relayer** smart contract is responsible for storing Merkle roots for blocks that were observed on the Vara chain. Before storing Merkle roots, the **Relayer** smart contract verifies received Merkle roots with the help of the **Verifier** smart contract.

The **Relayer** uses the `submitMerkleRoot` function to receive Merkle roots with the following details:

- **block_number**: the number of the block on the Vara chain
- **merkle_root**: Merkle root of transactions included in the block with corresponding block number
- **proof**: serialized plonk proof.

Upon successfully storing data about the block number and corresponding Merkle root, the **Relayer** smart contract will emit a MerkleRoot event.

It is important to note that anyone can submit a Merkle root because only validated Merkle roots will be stored in the **Relayer** smart contract.

Verifier smart contract: proof verification

The **Verifier** smart contract is responsible for verifying proofs. This is done with the help of the **PlonkVerifier** smart contract which is out of scope for this audit.

The **Verifier** uses the `verifyProof` function to verify the proof, where it processes the following details:

- **proof**: serialized plonk proof
- **public_inputs**: reduced public_inputs

This function will relay the message to the **Verify** function of the **PlonkVerifier** smart contract to finish the process of verification.

MessageQueue smart contract: verifies message origin

The main role of the **MessageQueue** smart contract is to verify that the received message originated from the Vara chain.

The **MessageQueue** uses the `processMessage` function to verify the origin of the message, where it processes the following details:

- **block_number**: block number of targeted Merkle root
- **total_leaves**: number of leaves in target Merkle root
- **leaf_index**: index of the leaf containing the message
- **message**: target message
- **proof**: Merkle proof of inclusion of leaf in targeted Merkle root

In this process, the **MessageQueue** smart contract will calculate the Merkle root for the message and validate that it corresponds to the Merkle root which is already stored in the **Relayer** smart contract for the same block number. If the proof is correct, the nonce of the received message will be stored in the smart contract and the message will be forwarded to the adequate message receiver, either **ERC20Manager** or **ProxyUpdater** (to be discussed later) smart contract.

Upon successful processing of the message MessageProcessed event is emited.

It is important to note that anyone can submit a message because all messages will be validated against previously stored Merkle roots in the **Relayer** smart contract.

ERC20Manager smart contract: unlocking/minting tokens on the Ethereum chain

The **ERC20Manager** smart contract processes messages received only from the **MessageQueue** smart contract.

The `processVaraMessage` function is used for unlocking/minting tokens on the Ethereum side, where it processes the following details:

- **sender**: the sender of the message on the Vara chain
- **payload**: message payload

Upon validating that the message sender is the VFT_Manager program from the Vara chain, in case the payload contains a reference to an Ethereum token, the required amount of the Ethereum tokens will be unlocked, while if it contains a reference to the Gear token, the required amount of tokens will be minted through the call to **ERC20GearSupply** smart contract.

Path Vara to Ethereum Integration

In this process, the interaction between the **Vara** network and **Ethereum** occurs as follows:

1. **Storing block numbers and corresponding Merkle roots**: As a first step and prerequisite for successful token bridging a valid Merkle root for the block containing the bridging transaction must be stored in the **Relayer** smart contract.
2. **Validating messages originating from the Vara chain**: In a separate transaction, a message contacting bridging data is sent to the Ethereum chain by calling the **MessageQueue** smart contract. **MessageQueue** will validate this message comparing it with the previously stored Merkle root for the corresponding block number that the message belongs to. If valid, message data will be forwarded to the **ERC20Manager** smart contract
3. **Validating sender and minting/unlocking tokens**: Message data received by the **ERC20Manager** smart contract will be checked if the message originated from VFT_Manager program and if so, required tokens, either Gear or Ethereum, will be unlocked or minted.

Ethereum to Vara Flow

The process of bridging tokens from **Ethereum** to **VARA** involves **ERC20ManagerBridgingPayment**, **BridgingPayment**, and **ERC20Manger** smart contracts.

BridgingPayment: collecting fees

The **BridgingPayment** smart contract is responsible for collecting bridging fees for transferring tokens from Ethereum to Vara chains. The smart contract enables setting the fee amount, updating the admin address that will collect the fee, and setting the underlying smart contract that will handle bridging requests.

The `deductFee` function is used for collecting the bridging fees. Upon calling this function the set amount of fee will be transferred from the function caller to the admin address and the FeePaid event will be emitted.

ERC20ManagerBridgingPayment: temporary token holder tokens

The **ERC20ManagerBridgingPayment** smart contract represents one of two entry points for transferring tokens from Ethereum to VARA chain. The smart contract initiates the fee payment process and temporarily holds tokens intended for bridging to VARA side.

The `requestBridging` function is used for initiating unlocking/minting tokens on the Ethereum side and temporary holding tokens, where it processes the following details:

- **token**: address of ERC smart contract for managing tokens
- **amount**: the number of tokens to be bridged
- **to:** token receiver on VARA chain

Upon receiving the bridging request, the **ERC20ManagerBridgingPayment** smart contract will forward the call to the **BridgingPayment** smart contract to facilitate payment of the fees. If payment is successful, calling the ERC20 smart contract located at **the token** address will transfer tokens from the account that originated the bridging request to the **ERC20ManagerBridgingPayment** smart contract. In the next step, **ERC20ManagerBridgingPayment** will give approval to the underlying **ERC20Manger smart** contract to manage the specified amount of tokens. **ERC20ManagerBridgingPayment** will then call `requestBridging` of **ERC20Manger** smart contract.

It is important to note that utilizing the **ERC20ManagerBridgingPayment** smart contract is optional. Users can bypass it and directly interact with the **ERC20Manger** to initiate the token transfer. In this case, the user will manually relay the request to the VARA chain.

ERC20Manager: burning/locking of tokens

The **ERC20Manager** smart contract performs the process of burning/locking specific tokens and also represents a second entry point into the bridging process in case the user wishes to bypass the fee payment**.** The `requestBridging` function is used for unlocking/minting tokens on the Ethereum side, where it processes the following details:

- **token**: address of ERC smart contract for managing tokens
- **amount**: the number of tokens to be bridged
- **to:** token receiver on VARA chain

Upon receiving the bridging request, the **ERC20Manager** smart contract will determine the type of tokens that are being bridged based on the **token** address and in case of Gear token transfer the request to the **ERC20GearSupply** smart contract for burning the tokens and in case of Ethereum token locking tokens at **ERC20Manager** smart contact address. Upon successful bridging request is performed `BridgingRequested` event will be emitted.

Path of the Ethereum to Vara Integration

- In this process, the interaction between the **Ethereum** network and **Vara** occurs as follows:

1. **Initiating bridging process**: There are two possibilities for initiating the bridging requests, one if the user chooses to pay the bridging fee and the other when the user manually relays the message. In the first case, the user will directly interact with the **ERC20ManagerBridgingPayment** smart contract while in the other case, the user will directly interact with the **ERC20Manager** smart contract.
2. **Fee payment**: In the case of using **ERC20ManagerBridgingPayment** defined amount of fee is collected.
3. **Locking/unlocking tokens**: Upon fee payment or as a result of a direct call **ERC20Manager** smart contract will perform token burning/locking based on the specified token address.

Upgrading smart contracts

The process of upgrading smart contracts is managed by **ProxyUpdater** and **ProxyContract** smart contracts.

ProxyUpdater: verifying the message origin and discriminator

The **ProxyUpdater** smart contract represents the entry point into the upgrade process through `processVaraMessage` function. The `processVaraMessage` function is used to verify the origin of the message on the VARA chain, the origin of the call on the Ethereum chain, and to determine the message discriminator value by processing the following parameters:

- **sender**: the sender of the message from the VARA chain
- **payload**: message payload

Upon receiving a call, `processVaraMessage` function will first validate that the function was called by **MessageQueue** smart contract. After that, the function will check that the sender of the message on the VARA chain matches the governance value. Both **MessageQueue** smart contract's address and governance value are initially stored at the **ProxyUpdater** smart contract at the time of deployment. Upon successful message verification, the function will determine the type of the message based on the initial byte of the payload. The message could be a request to update the address of underlying smart contracts implementation, to change the admin of the underlying smart contract or to change the value representing the governance. In the first two cases, the process will continue by calling the appropriate functions of **ProxyContract**, while the updated value of the governance is stored in the **ProxyUpdater** smart contract.

ProxyContract: updating the implementation and the owner

The **ProxyContract** extends the functionalities of the **Proxy** smart contract from the OpenZeppelin. The ProxyContracts performs the tasks of upgrading the implementation and updating the owner address through `upgradeToAndCall` and `changeProxyAdmin` functions. The `upgradeToAndCall` function has two parameters

- **newImplementation**: the address of the new underlying smart contract logic
- **data**: the initial call to be sent to newly deployed smart contract

Upon receiving a call, the `upgradeToAndCall` function will check if the caller of this function is a proxy admin, and if so, forward the call toward the `upgradeToAndCall` of OpenZeppelin Proxy smart contract with the same parameters.

The `changeProxyAdmin` function has only one parameter:

- `newAdmin` : the address of the new underlying proxy admin

Upon receiving a call, the `changeProxyAdmin` function will check if the caller of this function is a proxy admin, and if so, forward the call toward the `changeProxyAdmin` of OpenZeppelin Proxy smart contract with the same parameters.

Invariant Analysis

checkpoint-light-client

Invariant 1. Sync Committee set representation on checkpoint-light-client can not be changed to something that does not agree with the true sync committee, i.e. only a previous sync committee can approve the new sync committee.

Conclusion: This invariant holds. The sync committee is continuously updated, starting from an initial set and transitioning to subsequent sets, provided these are approved by the current set. This process is managed through the checkpoint-light-client using the `SyncUpdate::handle` function, which is invoked via the `sync_update::handle` mechanism.

The initial committee set is established using the `init()` function. This function employs the same verification logic as `sync_update::handle` to validate the first `sync_committee` and `finalized_header`. These values are loaded during initialization. At this stage, the finalized state and the initial committee set are derived directly from the update being processed.

The `finalized_header` from the update represents the finalized and already accepted state. It is validated by invoking the `sync_update::verify` function, which ensures the update's correctness. The only modification to the `finalized_header` is adjusting its `slot` to correspond to the previous slot, as no prior state exists during initialization.

This ensures that the sync committee updates and the finalized state transitions are handled securely and correctly, maintaining the integrity of the protocol.

Regular sync committee and header update is done through `ync_update::handle` as previously mentioned. In contrast to the setting the initial state, the regular state update relies on the existing state in the light client. The update guarantees sequential update with this check. This check is verifying whether the **gap between the epoch of the light client's current finalized header** and the **epoch of the new finalized header in the received update** exceeds a predefined maximum allowed gap (`MAX_EPOCHS_GAP`). If this condition is true, it indicates that the received update is **too far ahead** of the light client's current state, requiring a replay to ensure consistency. If this check passes then the `verify` function is called on the upcoming updates.

The `verify` function performs several key validations for the sync committee update in the light client process. After it verifies the temporal consistency of the update by ensuring that the slots associated with the **signature**, **attested header**, and **finalized header** follow the proper chronological order, it determines whether the current or next sync committee is applicable based on the update's `signature_slot` and ensures the update falls within the correct sync committee period. It then verifies quorum by checking that at least two-thirds of the sync committee members participated in the update, rejecting it if insufficient participation is detected. Then the function authenticates the aggregate signature using the public keys of the participating committee members, the `attested_header` , and the `signature_slot` . These validations ensure the sync committee update is legitimate, aligns with the light client's current state, and can be safely applied. The sync committee update depends if the merkle proof check if the given finalized header is definitely final according to the given `attested_header` . Each of these checks ensure the following connections:

- chronological
- sync committee signature
- header finalization

between the trusted light client state and the given update (`attested_header` , `finalized_header` `committee_update`) making it impossible to forge malicious update (any of its components) such to pass these checks in order to update the sync committee to a desired malicious set.

Invariant 2. Only finalized Ethereum headers (as confirmed by the sync committee) are accepted as valid. No unfinalized or potentially reorg-able headers are processed.

Conclusion: This invariant holds. The finalized Ethereum header is updated along with the sync committee update. These updates are done through the same handle functions in checkpoint-light-client.

As well as the sync committee, the finalized header is continuously updated, starting from an initial header and transitioning to subsequent headers, provided these are approved by the trusted sync committees set. This process is managed through the checkpoint-light-client using the `SyncUpdate::handle` function, which is invoked via the `sync_update::handle` mechanism.

The initial finalized header is established using the `init()` function. This function employs the same verification logic as `sync_update::handle` to validate the first `sync_committee` and `finalized_header` . These values are loaded during initialization. At this stage, the finalized state and the initial committee set are derived directly from the update being processed.

The `finalized_header` from the update represents the finalized and already accepted state. It is validated by invoking the `sync_update::verify` function, which ensures the update's correctness. The only modification to the `finalized_header` is adjusting its `slot` to correspond to the previous slot, as no prior state exists during initialization.

The `verify` function assesses whether the finalized header in the `sync_update` should replace the stored finalized header by performing a series of validations. It first ensures the chronological order of slots, verifying that the `signature_slot` and `attested_header.slot` are consistent with the `finalized_header.slot` . Next, it checks if the update's finalized header slot is more recent than the stored header, making it eligible for an update. The function then validates the finality proof by verifying the `finality_branch` against the attested header's state root to confirm that the update's finalized header aligns with the blockchain's Merkle structure. If all checks pass, the finalized header is updated to reflect the latest securely finalized checkpoint. This process ensures that the light client maintains an accurate and securely validated view of the blockchain's finalized state.

Invariant 3. Synchronization updates from Ethereum must reflect the most recent valid block headers and sync committee changes without skipping or duplicating any updates.

Conclusion: This invariant holds. As mentioned in the previous two invariant analysis the light client update is done continuously, starting from an initial header and transitioning to subsequent headers, provided these are approved by the trusted sync committees set. This is enforced with this check. This check is verifying whether the **gap between the epoch of the light client's current finalized header** and the **epoch of the new finalized header in the received update** exceeds a predefined maximum allowed gap (`MAX_EPOCHS_GAP`). If this condition is true, it indicates that the received update is **too far ahead** of the light client's current state, requiring a replay to ensure consistency. If this check passes then the `verify` function is called on the upcoming updates. Duplication of header and committee update is prevented here. This also enables independent update of header and committee.

Invariant 4. Error Handling Robustness: If invalid data or a failed proof verification attempt occurs, the checkpoint-light-client must reject the data and log an error without affecting its synchronization state.

Conclusion: This invariant holds. Every unsuccessful verification of the upcoming header and sync committee is ended with an error preventing the update with byzantine data.

ethereum_event_client

Invariant 1. Each event must be accompanied by a cryptographically verifiable proof of inclusion.

Conclusion: This invariant holds. Each event (`EthToVaraEvent`) to be checked is accompanied with a block inclusion proof and Merkle-PATRICIA proof. Both of these are checked against the current state of the checkpoint-light-client. This verification will be analyzed in the following two invariants.

Invariant 2. Proofs provided for event verification must align with the Ethereum chain's Merkle trie structure and be consistent with the current state of the checkpoint-light-client.

Conclusion: This invariant holds. Both proofs (block inclusion proof and Merkle-PATRICIA proof) must be aligned with the Ethereum chain because these are checked against the state of the checkpoint-light-client that is a light client representation of the Ethereum chain.

Invariant 3. The ethereum-event-client relies on the checkpoint-light-client to validate Ethereum blocks. It must reject any event from a block not verified as part of the canonical chain by the checkpoint-light-client.

Conclusion: This invariant holds. Ethereum-event-client uses checkpoint-light-client to perform the following two verifications on the given `EthToVaraEvent` :

- Block inclusion proof verification
- Merkle-PATRICIA proof verification

Prior to the verification current checkpoint-light-client state is obtained to act as a trusted state against which the verifications will be performed.

Block inclusion proof verification part of the code is validating the block inclusion proof by iteratively checking the integrity of block headers in reverse order, starting from a checkpoint hash. The `checkpoint` variable is a cryptographic hash representing the most recent known valid state of the blockchain. It is retrieved using `request_checkpoint` and corresponds to a specific slot in the blockchain. It acts as a starting point for validating a sequence of headers (`headers`) provided in the block inclusion proof. The goal is to ensure that the chain of headers leads from the given `checkpoint` to the block being verified.

Merkle-PATRICIA proof verification validates a Merkle-Patricia proof, which ensures that a specific Ethereum transaction receipt exists in the block's receipt trie. First, a memory-backed database (`memory_db`) is created using proof nodes provided in the inclusion proof. A `TrieDB` is constructed from the `receipts_root` and the proof nodes. Then, the encoded transaction index (`key_db`) and receipt (`value_db`) are looked up in the trie. If the trie contains the key and its value matches the expected receipt, the proof is valid; otherwise, the verification fails. This process guarantees the receipt's integrity and authenticity in the context of the block.

historical-proxy

Invariant 1. The historical-proxy must correctly route requests to the appropriate ethereum-event-client instance responsible for verifying the specific event or transaction.

Conclusion: The `historical-proxy` adheres to Invariant 1 by ensuring that requests are routed correctly to the appropriate `ethereum-event-client` instance responsible for verifying the specific transaction. This is achieved through the `redirect` method, which invokes the `endpoint_for` method to determine

the correct endpoint based on the specified `slot`. This design ensures accurate routing and maintains the integrity of the event verification process.

Invariant 2. The historical-proxy must maintain an accurate and up-to-date record of all ethereum-event-client program addresses, ensuring historical queries are directed to the correct client.

Conclusion: The `historical-proxy` maintains the state where each endpoint is stored as a pair: `slot` and `actor_id`, with `actor_id` representing the address of the `ethereum-event-client` program responsible for that slot. This design ensures accurate routing of historical queries to the correct client.

Invariant 3. Unauthorized changes to the historical-proxy program is not allowed. Only an admin can do this.

Conclusion: The `add_endpoint` method verifies the caller (`source`) against the `admin` field in the state to prevent unauthorized changes.

Invariant 4. The historical-proxy must ensure that events it processes are verified by the ethereum-event-client. It must not accept unverified or invalid data.

Conclusion: The `historical-proxy` ensures adherence to Invariant 4 by delegating event verification to the appropriate `ethereum-event-client` instance. Once the responsible endpoint for a specified slot is determined, the `historical-proxy` forwards the provided proofs to that endpoint for verification. It does not perform any verification itself but relies entirely on the endpoint to validate the data.

bridging-payment

Invariant 1. The bridging-payment program must always collect the exact fee specified for a bridging request before processing it. Requests without sufficient fees must be rejected.

Conclusion: The program is making sure the user which called `make_request` method is attached exactly the same amount of value for the message as specified in config fee parameter. If for some reason execution fails, the fee is returned to the user.

Invariant 2. Bridging requests must only be forwarded to the `vft-manager` after the fee has been successfully collected.

Conclusion: Similarly to the previous invariant, as program is making sure fees are paid, there is no possibility request can be forwarded to the `vft-manager` if fees are not paid as the program panics in that case.

Invariant 3. Every bridging request must emit a corresponding event that accurately reflects the status of the fee collection and request forwarding.

Conclusion: If forwarded request to the `vft-manager` is successfully finished, program emits `TeleportVaraToEth` event which means that user paid fees for the request to the relayer and relayer takes action from there.

Invariant 4. Only the `admin_address` is allowed to perform sensitive actions, such as:

- **Setting the fee (`set_fee`).**
- **Updating the VFT manager address (`update_vft_manager_address`).**
- **Configuring the program (`set_config`).**
- **Reclaiming accumulated fees (`reclaim_fee`).**

Unauthorized actors cannot modify critical program configurations or access fees.

Conclusion: All of the aforementioned methods are having `ensure_admin` check which makes sure only admin can change the state by calling that methods.

Invariant 5. The program must maintain an accurate and up-to-date `vft_manager_address`. Incorrect or stale `vft_manager_address` must not disrupt operations.

Conclusion: The responsibility for updating `vft_manager_address` relies completely on configured admin. The "official" `vft_manager_address` should be known to all bridge participants so that even admin changes the address to something arbitrary, it doesn't mean the rest of the actors in the flow will apply to that change.

Invariant 6. Bridging requests (`make_request`) require sufficient gas to:

- **Send the request to the VFT manager.**
- **Process the request.**
- **Return any required deposits or refunds.**

If insufficient gas is attached, the request must be rejected gracefully.

Conclusion: The config gas parameters are stored in the program so that before executing the message to `vft-manager` it is checked whether user will have sufficient gas. The check is not necessarily needed, since the request without enough gas will fail anyway due to the protocol, but could prevent user to spend the gas further as the request then fails immediately. Again, it's up to admin to insert the correct values so that the parameters represents real amount of gas needed. To sum up, if user doesn't have enough gas, the request will fail without any consequences.

vft-manager

Invariant 1. Token Supply Type Validation: Depending on the token type to be bridged certain operations can be performed:

- `Ethereum` **type tokens can be only minted and burned.**
- `Gear` **type tokens can only be locked or unlocked.**

Unsupported token supply type must not cause runtime errors.

Conclusion: This invariant holds.

When a token supply is located on Ethereum, tokens are minted/burnt on the Gear side and locked/unlocked on the Ethereum side. On the other hand, when a token supply is located on the Gear tokens are locked/unlocked on the Gear side and minted/burnt on the Ethereum side.

Token supply is modeled as an enum, holding Ethereum and Gear values.

For Token Supply Type validation in request bridging:

Vft-manager exposes Request bridging method. It covers Gear side of **request bridging** by handling **burn** for Ethereum token supply and **lock** for Gear token supply depending on the initiated transaction.

- Burn is handled by Extended vft client. When this is done, first, tokens are sent for burn by Vft-manager, a timeout is set, a reply hook handle is set, reply is awaited, and then, if everything is OK, a token deposit is made.
- Lock is handled by Extended vft client. When this is done, tokens are transferred from the sender to the Vft-manager's address, a timeout is set, a reply hook handle is set, reply is awaited, and then, if everything is OK, a token deposit is made.

Also, after a message is sent to bridge built-in, **Vft-manager** handles the error reply for each token supply type by **minting** burned tokens for Ethereum token supply and **unlocking** locked tokens for Gear token supply.

The above mentioned reply hook handle also decodes reply based on token supply value (all values are handled) and the message status.

To recover funds, Vft-manager provides a method (still, this method is not called from the code; assumption is that it has to be called from the front end;) that recovers funds that were stuck in the middle of the bridging. It handles both types of token supply by **minting** burned tokens for Ethereum token supply and **unlocking** locked tokens for Gear token supply.

For Token Supply Type validation in submit receipt.

Vft-manager exposes the Submit receipt method (called from HistoricalProxyService). It withdraws deposited funds from Vft-manager for a received transaction receipt from Ethereum ERC20 manager and an unprocessed transaction. **Vft-manager** covers Gear side of transaction initiated from Ethereum side by handling **mint** for Ethereum token supply and **unlock** for Gear token supply (both token supply types are handled).

- Mint is handled by Extended vft client. When this is done, first, tokens are sent for minting by Vft-manager, a timeout is set, a reply hook handle is set, reply is awaited, and then, if everything is OK, a token withdrawal is made.
- Unlock is handled by Extended vft client. When this is done, tokens are transferred from the Vft-manager's address to the recipient's address, a timeout is set, a reply hook handle is set, reply is awaited, then, if everything is OK, a token withdrawal is made.

Also, the above mentioned reply hook handle decodes the reply based on the token supply value (all values are handled) and the message status.

Findings:

- Fix documentation comment of Token supply type for Gear token supply type.

Invariant 2. Invalid `vara_token_id` must not cause runtime errors and transactions using invalid token addresses (Vara and Erc20 tokens).

Conclusion: This invariant holds.

TokenMap (a mapping of vara token address, Ethereum token address and supply type) is populated by an admin and persisted in Vft-manager's state. TokenMap supports only a single mapping of vara_token_id (i.e. vara token address) to a Token Supply type value (Ethereum or Gear) and Vara <> Ethereum address mapping, which enables representation of the token on one chain to correspond to its existence on the other chain. In this way TokenMap controls allowed token transfer as Gear side applies checks of the source and destination transaction token regardless of the side that initiated a transaction:

- A check is done in the Submit receipt, using the TokenMap to find vara_token_id (address) based on the supplied Ethereum token id (eth_token_id) from the decoded receipt. This check returns a Vara error if the address is not found.
- A check is done in the Request bridging, using the TokenMap to find eth_token_id (address) based on the supplied vara_token_id from the request. This check returns an Ethereum error if the address is not found.

As for the application of vara_token_id in the Vft-manager, since allowed vara_token_ids are defined by admin and persisted in the TokenMap, Vft-manager does not create a new vara_token_id but propagates only the valid/checked vara_token_id supplied by submitted receipt or bridging request.

Invariant 3. Message Tracking Integrity:

- **Every request is tracked using a unique `msg_id` and stored with the associated transaction details and status.**
- **The `msg_tracker` must always maintain an accurate record of the message's lifecycle:**
  - **request_bridging operation:**
    - **Initializing with `SendingMessageToDepositTokens`.**
    - **Updating to `SendingMessageToBridgeBuiltin` or `SendingMessageToReturnTokens` based on the outcome.**
    - **Updating to `TokenDepositCompleted`, `BridgeResponseReceived` or `TokensReturnComplete` based on the outcome.**
  - **submit_receipt operation:**
    - **When message to withdraw tokens is sent status needs to be `SendingMessageToWithdrawTokens.`**
    - **Update to `TokenWithdrawComplete` when a reply is received for token withdrawal.**

Loss of message status or incorrect tracking can lead to orphaned (incompletely processed) transactions.

Conclusion: This invariant holds, as a message transitions to the states in order that reflects `request_bridging` and `submit_receipt` processes.

Vft-manager initializes separate message trackers for request bridging and submit receipt. Each MessageTracker is a mapping of message id to message info (status and tx details). Message status keeps track of MessageTracker state machine, supporting two states (`SendingMessageToWithdrawTokens` and `TokenWithdrawComplete`) for submit receipt, and six states (`SendingMessageToDepositTokens`, `TokenDepositCompleted`, `SendingMessageToBridgeBuiltin`, `BridgeResponseReceived`, `SendingMessageToReturnTokens` and `TokensReturnComplete`) for request bridging.

For `request_bridging` operation:

A message is created in the `request_bridging` operation: `msg_id` is generated by gstd library, tx details are populated from input parameters, while a message status is set to `SendingMessageToDepositTokens`. The message transits to `TokenDepositCompleted` after burn/lock is finished, which can be either of `TokenDepositCompleted(true)` and `TokenDepositCompleted(false)`.

- `TokenDepositCompleted(false)` ends processing with error, while
- `TokenDepositCompleted(true)` transitions to `SendingMessageToBridgeBuiltin`.

Further, from `SendingMessageToBridgeBuiltin` the message transitions to either `BridgeResponseReceived(Nonce)` or `BridgeResponseReceived(None)` when builtin returns a reply.

- `BridgeResponseReceived(Nonce)` ends processing successfully; message is deleted from tracker and Nonce is propagated. This is the only message state that leads to a message deletion in the code.

- `BridgeResponseReceived(None)` leads to error that transitions the message to `SendingMessageToReturnTokens` to undo depositing i.e mint/unlock which results in either `TokensReturnComplete(true)` or `TokensReturnComplete(false)`. There is also a possibility for mint/unlock to fail before, i.e. due to e.g. `SendFailure`. In that case the state transitions to `BridgeResponseReceived(None)`, as the state is reverted because of panic. There is also a possibility for mint/unlock to fail due to a Timeout error. In that case `SendingMessageToReturnTokens` is saved, the state is reverted because of panic, and the state would be eventually overridden by the `reply_hook` (assuming that a panic does not happen within the hook). If the panic happens in the hook the state is reverted to `BridgeResponseReceived(None)` or `SendingMessageToBridgeBuiltin` (depending on the failure type).

If builtin fails with error before inner await (leaf future), `SendingMessageToBridgeBuiltin` transitions to `SendingMessageToReturnTokens` continuing like described above.

A message is removed from a message tracker only when first four statuses are passed successfully (`SendingMessageToDepositTokens`, `TokenDepositCompleted(true)`, `SendingMessageToBridgeBuiltin` and `BridgeResponseReceived(some nonce)`) meaning that depositing was done and builtin reply was ok, otherwise the message is left in the message tracker.

For `submit_receipt` operation:

A message is created in the `submit_receipt` operation: `msg_id` is generated by gstd library, tx details are populated from decoded receipt while a message status is set to `SendingMessageToWithdrawTokens`. The message transits to `TokenWithdrawComplete` after mint/unlock is finished without errors, which can be either of `TokenWithdrawComplete(true)` and `TokenWithdrawComplete(false)`. If an error is thrown before inner await, the state is reverted and message is not saved. If an error is thrown after inner await, the state is saved to `SendingMessageToWithdrawTokens`, the panic is raised, reverting the state and not saving the message and eventually the `reply_hook` will try to update non-existing message.

A message is added to a message tracker in `submit_receipt` method.

A message is never removed from the message tracker by the code, during message processing. It can be removed only by an explicit call to the MessageTracker.

Findings:

- User may not be able to revert deposited tokens
- Unnecessary retention of message info in status tracking
- Message Tracker / Message Statuses Redundancy

Invariant 4. Token Operation Rollback on Failure on Vara chain (does):

- **If the request to the bridge built-in actor fails:**
  - **Tokens must be refunded to the sender.**
  - **The refund operation (minting or unlocking) must match the original token supply type.**
- **Rollbacks must handle partial state changes to ensure no user loses funds.**

No tokens can remain burned/locked without completion of the bridging process or refund.

Conclusion: This invariant holds partially, as the funds may stay minted/unlocked when the automatic refund process fails. A manual recovery process is provided via separate call.

As the bridge built-in always replies on submitted request (guaranteed by the Gear protocol) it either returns nonce or none. When none is returned, the request fails with an error. In that case, the message enters a refund process i.e. mint/unlock based on the original token supply type. The mint/unlock operation may fail as extended-vft returns boolean value. In case the operation fails (false is returned) the process ends with an error returned and a message in the message tracker having status `TokensReturnComplete(false)`. The only way to mint/unlock is to manually run recovery by calling `handle_request_bridging_interrupted_transfer` which may again succeed or fail.

Invariant 5. Gas Efficiency:

1. **The bridging process and rollback operations rely on sufficient gas being attached.**
2. **Insufficient gas must gracefully handle errors, ensuring no funds remain inaccessible.**

Requests with insufficient gas must not disrupt the system or leave funds in limbo.

Conclusion: The invariant holds.

Gas efficiency:

1. Sufficient gas:

- Bridging process - burn/lock and sending to bridge builtin rely on the function `gstd::msg::send_bytes_with_gas_for_reply` which among other parameters specifies: `gas_limit` - determines how much gas is allocated for the execution of the recipient's program, and `reply_deposit` - determines a deposit for storing the reply message, if any, returned by the recipient. These values are read from vft-manager's config. `gas_limit` is `gas_for_token_ops` or `gas_to_send_request_to_builtin` depending on the operation, while `reply_deposit` is `config.gas_for_reply_deposit`. Config is initialized when vft-manager is instantiated and can be updated by an admin. Result of `gstd::msg::send_bytes_with_gas_for_reply` has to be decoded.
- Rollback - Similarly, mint/unlock relies on `gstd::msg::send_bytes_with_gas_for_reply`, where `gas_limit` is `gas_for_token_ops` config value.

Team provided information that current deployment uses a gas limit that is 5x to 10x the size of the actual gas limit spent on the operation.

2. Insufficient gas:

- Error handling - when insufficient gas is provided then `gstd::msg::send_bytes_with_gas_for_reply` returns an error mapped to `Error::SendFailure`. For burn/lock this means that deposit is not made and the message status is not relevant. For send to built-in it means that the message status is `SendingMessageToBridgeBuiltin` and transitions to the next state for return of funds. For mint/unlock when `Error::SendFailure` happens, the code panics caused by expect and the message status is reverted to status from the previous successful leaf future and the process ends.

Invariant 6. Event Emission:

A successful bridging request must emit the `BridgingRequested` event with:

- **The `nonce`.**
- **The `vara_token_id`, `amount`, `sender`, and `receiver`.**

Conclusion: The invariant holds.

Vft-manager emits `BridgingRequested` with `nonce, vara_token_id, amount, sender, receiver` values upon successful request bridging.

Invariant 7. Bridging request replay protection:

Interrupted or failed bridging requests must only retry token refunds under specific conditions:

1. **Message status indicates incomplete or interrupted bridging.**
2. **The refund operation must not run for already-completed transactions.**

No double refunding or repeated operations are allowed.

Conclusion: This invariant holds.

Generally, each `request_bridging` call is a new request/transaction with a unique message id identifying the message having the details of the initiated transaction, and tracking its status during processing.

For a message status condition:

Automatic token refunds happen within request bridging processing when request to built in ends with `BridgeResponseReceived(None)` status or built-in fails before reply is received in which case status is `SendingMessageToBridgeBuiltin`. The status transitions to `SendingMessageToReturnTokens` and mint/unlock is tried.

On demand token refunds is initiated with `handle_interrupted_transfer` and it currently allows refund when a message is in `TokenDepositCompleted(true)`, `BridgeResponseReceived(None)` and `TokensReturnComplete(false)` statuses.

For already completed transactions:

When a transaction is completed in `request_bridging` a corresponding message is deleted from the message tracker.

Invariant 8. Submit receipt replay protection - message status:

Interrupted or failed submit receipts must only retry token withdrawal under specific conditions:

1. **Message status indicates incomplete or interrupted submit receipt.**
2. **The withdrawal operation must not run for already-completed transactions.**

No double funding or repeated operations are allowed.

Conclusion: This invariant holds.

Every submit_receipt corresponds to a single Ethereum transaction. Each transaction is added to transaction storage, before mint/unlock happens protecting another attempt of handling the same Ethereum transaction. If mint/unlock pass without errors then, corresponding transaction stays in the storage. Otherwise, expect will cause panic and the storage operation will be reverted.

For a message status condition:

There is no automatic withdrawal process. If withdrawal (mint/unlock) fails for any reason (for example, an early exit due to `SendFailure`, `ReplyTimeout`, etc. or failed reply `TokenWithdrawComplete(false)`, etc.) panic happens and storage changes are retrieved.

For already completed transactions:

Withdrawal is not possible for already completed transactions as they are kept in transaction storage.

Invariant 9. Historical Proxy Authorization:

1. **Only the `historical_proxy_address` program is authorized to call the `submit_receipt` function.**
2. **Any other sender must trigger an `Error::NotHistoricalProxy`.**

Unauthorized access to the method must be strictly prevented.

Conclusion: This invariant holds.

`submit_receipt` code checks sender's address allowing only HistoricalProxy to make `submit_receipt` calls.

Invariant 10. Submit receipt transaction History Management:

1. **The `TRANSACTIONS` set maintains up to `TX_HISTORY_DEPTH` entries.**
2. **Older transactions are removed (`pop_first()`) as newer ones are added, ensuring the set remains within size limits.**

The history is maintained efficiently to prevent memory overflow.

Conclusion: This invariant holds.

`submit_receipt` code takes care that transaction store (i.e. `TRANSACTIONS` set) keeps up to `TX_HISTORY_DEPTH` entries. A new transaction is added only when it is younger than the oldest one from the transactions store. If the store is full the oldest one is removed and the new one is added.

Invariant 11. Submit receipt reply protection - transactions:

Each Ethereum transaction is uniquely identified by a `(slot, transaction_index)` pair.

- **If a transaction is already in the `TRANSACTIONS` set, it triggers an `Error::AlreadyProcessed`.**
- **Transactions older than the `TX_HISTORY_DEPTH` limit and earlier than the oldest tracked transaction are rejected with `Error::TransactionTooOld`.**

No double processing or transactions outside the allowed depth are permitted.

Conclusion: This invariant holds. There is no double processing nor transactions outside of the allowed depth.

`AlreadyProcessed` error is thrown when transaction is already stored in the transaction storage. Also, if the transaction is older than the oldest transaction in the storage when the storage is full than `TransactionTooOld` error is thrown. If the storage is not full or the transaction is younger than the oldest one than the transaction is stored.

Invariant 12. Handling Interrupted Transfer - submit_receipt:

1. **Failed requests due to insufficient gas or temporary errors must retry token minting/unlocking if message status is `TokenWithdrawComplete(false)`.**
2. **Transactions marked as complete or with unexpected statuses must not retry.**

Retrying is allowed only for incomplete transfers, preventing redundant or conflicting operations.

Conclusion: This partially holds as retry has to be done manually.

When an error is thrown during mint/unlock a message status is set to `TokenWithdrawComplete(false)` (reply was decoded as false, and mapped to `FailedMessage`) or if the mint/unlock is exited before reply (e.g. `SendFailure`) status stays `SendingMessageToWithdrawTokens` or if reply fails `ReplyFailure` the status stays

`SendingMessageToWithdrawTokens` then panic happens which restores the state clearing transaction store from the current transaction and the message is cleared.

As there is no automatic retry of mint/unlock manual request for `handle_interrupted_transfer` has to be done, which only retries mint/unlock for `TokenWithdrawComplete(false).`

Findings:

- Unreachable condition for successfully handling interrupted transfers

ethereum common

Invariant 1. All relevant and necessary structures, data, and functions required for accurately representing the Ethereum-based chain (Beacon) within the bridge on Vara must adhere strictly to Ethereum standards as defined in its Go implementation. This ensures consistency, compatibility, and correctness in the interaction between the two chains.

Conclusion: The invariant holds. Upon review, it is evident that the Rust implementation adheres to Ethereum standards, despite minor differences stemming from variations in data structures and naming conventions. The functionality aligns with Ethereum's requirements, and all necessary structures, data, and functions are implemented correctly. Comments across both implementations confirm that the intended operations are equivalent, with no indications of missing or incorrect elements in the Rust implementation.

Ethereum Smart Contracts

Invariant 1. Merkle roots are submitted to the Relayer smart contract with complete and sound zk proofs, i.e., the gnark verifier rejects invalid proofs and accepts valid ones.

Threat 1: Proofs not representing a valid Vara state are stored.

Conclusion: This threat does not hold. Before storing Merkle roots for any submitted block, Merkle roots are verified through the PlonkVerifier smart contract.

Threat 2: The verifier rejects valid proofs.

Conclusion: This threat does not hold. There is a possibility that in the case of a fork in the VARA chain, a valid Merkle root originating from the fork might be submitted to the Relayer smart contract and that that Markle root will get stored, before the arrival of the “valid” main branch Merkle root. In case of two valid Merkle roots, the Relayer smart contract will go into an emergency state holding further transfers until the issue is resolved.

Invariant 2. The Relayer smart contract is not prone to replay attacks, i.e. accepting the same message related to block and Merkle root will have different or unexpected outcomes

Threat: Accepting multiple messages with the same data about the block number and Merkle root might block the bridge.

Conclusion: This threat does not hold. Any actor can submit the block number and corresponding Merkle root. If the Merkle root is valid that data will be stored/overwritten without causing any issues in the functionality of the bridge.

Invariant 3. The Relayer smart contract can recover from the case in which different valid Merkle roots are sent for the same block.

Threat: Accepting multiple messages with the same data about the block number and Merkle root might block the bridge.

Conclusion: This threat holds. There is a possibility that in the case of a fork in the VARA chain, a valid Merkle root originating from the fork might be submitted to the Relayer smart contract and that that Markle root will get stored, before the arrival of the “valid” main branch Merkle root. In case of two valid Merkle roots, the Relayer smart contract will go into an emergency state holding further transfers until the issue is resolved. If/when the issue is resolved there is currently no possibility for the Relayer smart contract to exit the emergency state. For now the development team is ok with this behavior because this means that the bridge is compromised and that no further actions are sensible.

Invariant 4. The MessageQueue smart contract rejects replay attacks.

Threat: A VaraMessage with valid proof submitted to the Relayer smart contract is processed more than once in the MessageQueue smart contract.

Conclusion: This threat does not hold. Although the MessageQueue smart contract will accept requests to process messages from any source, the VaraMessage structure declares a nonce field. Upon passing all the requirements, MessageQueue smart contract will flag the required nonce as processed thus disabling the possibility of executing the same request twice.

Invariant 5. The MessageQueue smart contract processes only messages that are verified against a trusted Merkle root.

Threat: The MessageQueue smart contract processes VaraMessages without verifying whether they match the trusted Merkle root.

Conclusion: This threat does not hold. Before processing VaraMessages, each message is checked against Merkle roots previously stored in the Relayer smart contract thus validating that they indeed originated from the valid VARA chain transaction.

Invariant 6. The MessageQueue smart contract is resilient to invalid proofs, i.e. invalid proofs do not block further processing of valid ones.

Threat: A VaraMessage with invalid proof submitted to the MessageQueue smart contract compromising liveliness, i.e. it can block the subsequent submission of a valid proof for a message with the same nonce.

Conclusion: This threat does not hold. If the proof is not valid the VaraMessage will not get processed and the nonce of that message will not be stored in the MessageQueue smart contract thus enabling for the valid message with the same nonce to be processed.

Invariant 7. The ERC20Manager smart contract rejects unvalidated VaraMessages.

Threat: A VaraMessage that has not been validated will be accepted for processing by ERC20Manager

Conclusion: This threat does not hold. The ERC20Manager smart contract will accept VaraMessages only from the MessageQueue smart contract by checking who is the message sender and comparing it to the address of the MessageQueue smart contract that is set during the deployment of the ERC20Manager smart contract.

Invariant 8. The ERC20Manager smart contract rejects replay attacks.

Threat: A VaraMessage that was once processed will be accepted again for processing by ERC20Manage

Conclusion: This threat does not hold. The ERC20Manager smart contract will accept VaraMessages only from the MessageQueue smart contract. As the MessageQueue smart contract is resilient to replay attacks and since it is the only source of VaraMessages for the ERC20Manager smart contract, it makes the ERC20Manager replay resistant too.

Invariant 9. The ERC20Manager smart contract rejects VaraMessages that were badly formatted.

Threat: A VaraMessage that carries badly formatted VaraMessage will be accepted for processing by ERC20Manager

Conclusion: This threat does not hold. Upon receiving the payload from VaraMessage, sent by the MessageQueue smart contract, the ERC20Manager smart contract checks the required length of the payload to verify that its format matches the expected format for the address receiver, address token, and uint256 amount.

Invariant 10. The ERC20Manager smart contract rejects VaraMessages that VFT Manager did not send.

Threat: A VaraMessage that was not sent by VFT message will be accepted for processing by ERC20Manager

Conclusion: This threat does not hold. Upon receiving the sender from the VaraMessage, sent by the MessageQueue smart contract, the ERC20Manager smart contract will verify that the sender matches the address of VFT Manager program from the VARA chain that is set during the deployment of the ERC20Manager smart contract (The process of setting the sender value is part of Bridging Invariant 5).

General bridge invariants

Invariant 1. Atomicity: Transfers across the bridge must either complete fully or not at all.

Conclusion: This invariant does not hold.

Generally, bridge programs do not support timeouts throughout the flows and rollback mechanisms are prone to errors (see previous findings).

Bridging payment uses a timeout when sending a bridging request to the Vft-manager. If the request fails, a fee is refunded to the sender.

Vft-manager's request_bridging use timeouts when handling burn/lock, sending to built-in and mint/unlock. If request bridging is not complete automatic refund process is run. If the automatic process fails, there is an alternative, a manually run process for handling interrupted transfers.

Vft-manager's submit_receipt use timeouts when handling mint/unlock. If the process fails, there is an alternative, a manually run process for handling interrupted transfers.

Historical proxy call to Vft-manager's submit_receipt does not use timeout.

Ethereum-light-client does not use a time out when requesting checkpoints from the checkpoint-light-client.

Checkpoint-light-client does not use timeouts in verifying sync committee signature (example).

Invariant 2. Reply protection: Retried or duplicated operations must not result in double transfers or inconsistent states.

Conclusion: This invariant holds.

Reply protection is applied in both request bridging and submit receipt.

Each `request_bridging` call is a new request/transaction with a unique message id identifying the message having the details of the initiated transaction, and tracking its status during processing.

Every `submit_receipt` corresponds to a single Ethereum transaction. Two mechanisms are applied for reply protection. The first one includes a transaction storage (i.e. transaction history) where each transaction is added, which prevents handling of a transaction if already present or too old. The second one relies on message ids that track processing of the initiated transactions.

Invariant 3. Token Supply Preservation: The total supply of a token must remain unchanged when mapped across chains.

Conclusion: This invariant partially holds. This invariant is closely linked with the first two General Bridge invariant. While the reply protection invariant hold based on the conclusions of the corresponding invariants of the responsible components on each of the chains the invariant regarding the atomicity of the transfers across the chain does not hold. Even though the development team finds acceptable having no automatic revert mechanism and that if the necessity arises that they will introduce necessary mechanisms through bridge protocol upgrade by the governance or an admin the lack of such mechanism means that this has a high probability of happening. This means that due to the fact that a transaction is stuck an imbalance in the token supply across the chains will happen.

Invariant 4. Liveness: The bridge must remain operational, and transactions should not get stuck indefinitely.

Conclusion: This invariant partially holds. Its fulfillment is closely linked to the first invariant in this group. The absence of revert and timeout mechanisms means that transactions can remain unprocessed and stuck in cases where, for example, the relayer is unavailable or a transaction is not accepted on the destination chain. While revert mechanisms exist on each chain, they are ineffective when a transaction has not yet been processed on the destination chain.

As a result, individual transactions may become stuck; however, this does not prevent users from submitting new transactions that can successfully process. This ensures that the bridge continues functioning despite issues with specific transactions. Additionally, the bridge will completely halt in the event of an emergency stop on the Ethereum side. No mechanism has been introduced for returning to normal operation after such a stop. However, the development team finds this acceptable, as an emergency stop indicates a critical issue with the bridge itself.

Findings

Title	Type	Severity	Status
Vulnerabilities in burn mechanism and exploitation via vft-manager	PROTOCOL IMPLEMENTATION	CRITICAL	RESOLVED
Split transactions for allowance and bridging introduce exploitation risk	PROTOCOL IMPLEMENTATION	HIGH	RESOLVED
User may not be able to revert deposited tokens	IMPLEMENTATION	MEDIUM	RESOLVED
Unnecessary retention of message info in status tracking	IMPLEMENTATION	LOW	ACKNOWLEDGED
Unreachable condition for successfully handling interrupted transfers	IMPLEMENTATION	LOW	RESOLVED
Message Tracker / Message Statuses Redundancy	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED
No validation of config inputs	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED

Vulnerabilities in burn mechanism and exploitation via vft-manager

Project	Gear 2025: Q1: Gear Bridges
Type	PROTOCOL IMPLEMENTATION
Severity	CRITICAL
Impact	HIGH
Likelihood	HIGH
Status	RESOLVED
Issue(s)	Vulnerabilities in burn mechanism and exploitation via vft-manager

Involved Artifacts:

- gear-programs/app/src/services/request_bridging/mod.rs

Description:

The burn functionality in the `extended-vft` implementation on Vara differs from Ethereum's ERC20 standards by allowing addresses with burn privileges to burn an unlimited amount of tokens from any user without requiring prior allowance or approval. This issue exposes users to unauthorized actions against their funds.

The situation is further complicated by the design of the `request_bridging` method in `vft-manager`, which allows an arbitrary sender address to be specified without verification. While this open design is intentional and expected to fail unauthorized actions due to allowance checks, the absence of allowance validation in the burn mechanism undermines this protection. This creates a significant security risk when these two components are combined.

Problem Scenarios:

A malicious actor could exploit the interaction between the unrestricted burn functionality and the `request_bridging` method to steal funds from unsuspecting users. Here's an example attack scenario:

1. The malicious actor calls the `request_bridging` method and specifies a victim's address as the sender.
2. Because the `vft-manager` does not verify the sender, it processes the request.
3. The `vft-manager` invokes the burn functionality, which allows the malicious actor to burn tokens from the victim's account without requiring any prior allowance.
4. The equivalent value of the burned tokens is then bridged to Ethereum and transferred to an address controlled by the malicious actor.

In this scenario, the malicious actor effectively steals the victim's funds by exploiting the combination of unrestricted burn permissions and the open design of `request_bridging`.

Recommendation:

To mitigate this risk, the burn mechanism in `extended-vft` should be updated to enforce allowance checks, ensuring that tokens can only be burned with explicit user approval. This change aligns the implementation with ERC20 standards and eliminates the core vulnerability.

Status:

This issue has been resolved through https://github.com/gear-tech/gear-bridges/pull/268 .

Split transactions for allowance and bridging introduce exploitation risk

Project	Gear 2025: Q1: Gear Bridges
Type	PROTOCOL IMPLEMENTATION
Severity	HIGH
Impact	HIGH
Likelihood	MEDIUM
Status	RESOLVED
Issue(s)	Split transactions for allowance and bridging introduce exploitation risk

Involved Artifacts:

- gear-programs/app/src/services/request_bridging/mod.rs

Description:

In the current design on Vara, users first grant allowances to `vft-manager` and then execute bridging operations through either the `bridging-payment` program (to pay relayer fees) or a direct call to `vft-manager` (requiring manual relay of the message to Ethereum). Unlike Ethereum's design, where allowance and transfer requests are atomic and executed within the same transaction, this approach separates the two steps into distinct transactions.

This separation creates a potential risk window between granting allowance and submitting the bridging request. During this time, an attacker could exploit the granted allowance by specifying the granting user as the sender and transferring funds to an arbitrary Ethereum address.

Problem Scenarios:

A practical example of this vulnerability:

1. A user grants allowance to `vft-manager` for a specified token amount.
2. Before the user submits their bridging request, an attacker learns about the allowance (e.g., by monitoring transactions or allowances on-chain).
3. The attacker submits a `request_bridging` call to `vft-manager`, specifying the user's address (which granted the allowance) as the sender.
4. Since the allowance is valid, `vft-manager` processes the burn/lock and bridges the funds to Ethereum. The attacker specifies their own address on Ethereum as the recipient, effectively stealing the victim's funds.

Recommendation:

To mitigate this risk, a more robust design should be considered to safeguard allowances. Combining the allowance grant and the bridging request into a single atomic operation would provide the most secure solution, ensuring allowances are immediately consumed and cannot be exploited.

If combining the two steps is infeasible, implementing additional safeguards, such as restricting allowances to the original granting user's transactions or limiting their validity to a short duration, could significantly reduce the risk.

Status:

This issue has been resolved through https://github.com/gear-tech/gear-bridges/pull/268 .

User may not be able to revert deposited tokens

Project	Gear 2025: Q1: Gear Bridges
Type	IMPLEMENTATION
Severity	MEDIUM
Impact	HIGH
Likelihood	LOW
Status	RESOLVED
Issue(s)	User may not be able to revert deposited tokens

Involved Artifacts:

- gear-programs/app/src/services/request_bridging/mod.rs

Description:

The `bridging_request` revert mechanism does not currently support refunding deposited funds when the process concludes with a status of `SendingMessageToBridgeBuiltin`. Although deposits are successfully made, it is still possible for the process to end in this status, despite the assumption that such a scenario would not occur.

Problem Scenarios:

In `request_bridging`, if the burn/lock operation completes successfully, the message status is set to `TokenDepositCompleted(true)` and subsequently transitions to `SendingMessageToBridgeBuiltin`.

If the call to `builtin` encounters a failure (e.g., `ReplyFailure` error), the `handle_reply_hook` is not executed, but the `await` operation completes, and the state is saved with the last set status, i.e., `SendingMessageToBridgeBuiltin`.

Following this, the status transitions to `SendingMessageToReturnTokens`, and the mint/unlock operation is initiated.

If this operation fails due to an error (e.g., `SendFailure`, `ReplyTimeout`), the expect statement within mint/unlock will trigger a panic, reverting the status from `SendingMessageToReturnTokens` back to `SendingMessageToBridgeBuiltin`, effectively terminating the process.

As a result, users may be unable to revert their funds, as `handle_interrupted_transfer` does not process the `SendingMessageToBridgeBuiltin` status.

Recommendation:

Review the usage of the `SendingMessageToBridgeBuiltin` status and consider updating `handle_interrupted_transfer` to account for this scenario. Ensure that any modifications do not introduce unintended behavior.

Status:

This issue has been resolved through https://github.com/gear-tech/gear-bridges/pull/275 .

Unnecessary retention of message info in status tracking

Project	Gear 2025: Q1: Gear Bridges
Type	IMPLEMENTATION
Severity	LOW
Impact	LOW
Likelihood	MEDIUM
Status	ACKNOWLEDGED
Issue(s)	Unnecessary retention of message info in status tracking

Involved Artifacts:

- gear-programs/vft-manager/app/src/services/request_bridging/bridge_builtin_operations.rs
- gear-programs/vft-manager/app/src/services/request_bridging/msg_tracker.rs
- gear-programs/vft-manager/app/src/services/submit_receipt/msg_tracker.rs

Description:

The message tracking system retains message information unnecessarily after certain processes are completed. This retention serves no functional purpose, as the status cannot progress further. In contrast, during the normal "happy path", the message info is deleted once the process completes.

Notably, a `TODO` comment in the code states: *"Remove completed messages from tracker,"* indicating awareness of the issue but leaving it unresolved.

Problem Scenarios:

- **Automatic revert in** `request_bridging`: When a revert (e.g., refunding tokens) is successfully executed within the `request_bridging` method, the message info persists even though the status cannot change further.
- **Manual execution of** `handle_interrupted_transfer`: After a user successfully executes `handle_interrupted_transfer` following a failure in either `request_bridging` or `submit_receipt`, the message info remains stored, despite the resolution being complete.
- **Successful completion of** `submit_receipt`: When `submit_receipt` is immediately and successfully completed, the message info is not deleted even though no further processing is required.

Recommendation:

Ensure that message info is deleted once the status cannot progress further, including after successful execution of `handle_interrupted_transfer` (both automatic and manual cases) and after the immediate successful completion of `submit_receipt`.

Unreachable condition for successfully handling interrupted transfers

Project	Gear 2025: Q1: Gear Bridges
Type	IMPLEMENTATION
Severity	LOW
Impact	LOW
Likelihood	LOW
Status	RESOLVED
Issue(s)	Unreachable condition for successfully handling interrupted transfers

Involved Artifacts:

- gear-programs/vft-manager/app/src/services/submit_receipt/token_operations.rs
- gear-programs/vft-manager/app/src/services/submit_receipt/mod.rs

Description:

If `submit_receipt` call does not pass, it is not possible to retry it using `handle_interrupted_transfer` as message status never ends in `TokenWithdrawComplete(false)` due to raised panic. Still another `submit_receipt` can be made as transaction storage was reverted.

Problem Scenarios:

In `submit_receipt`, when an error is thrown during mint/unlock a message status is either set to `TokenWithdrawComplete(false)` (when reply is decoded to false) or if the mint/unlock is exited before reply is handled (e.g. SendFailure, ReplyTimeout) status stays `SendingMessageToWithdrawTokens`, then panic happens due to an error. As it was explained by the team, the panic restores the state. As there is no automatic retry of mint/unlock, a manual request for `handle_interrupted_transfer` has to be done, which only retries mint/unlock for `TokenWithdrawComplete(false)` status.

When transitioned to `TokenWithdrawComplete(false)`, as an error is thrown and expect panics, the status is restored:

1. If it is restored, then the status is no longer `TokenWithdrawComplete(false)` and attempt to `handle_interrupted_transfer` will never execute mint/unlock as it only handles `TokenWithdrawComplete(false)`.
2. If the error is thrown before reply, than the status stays `SendingMessageToWithdrawTokens`, and when panic happens the status will be restored:
    a. In any case, regardless of the message status reversion (`SendingMessageToWithdrawTokens` or empty), if the message stays in the tracker,

`handle_interrupted_transfer` will never execute mint/unlock as it only handles `TokenWithdrawComplete(false)`.

3. Upon calling `submit_receipt`, tx is saved in the storage as a pair (slot, tx_index), meaning that it is allowed to enter the `submit_receipt` only once for the same transaction if the tx store state is not reverted upon panic.
    a. If everything is reverted on panic, including the message state (even the message), and tx store, then `handle_interrupted_transfer` will never serve its purpose (`TokenWithdrawComplete(false)` is never reached) and the only way to apply transaction is to call `submit_receipt` again.
    b. If tx store is not reverted then next `submit_receipt` attempts will fail. Then only way to apply receipt is to call `handle_interrupted_transfer` and assume that the status is `TokenWithdrawComplete(false)` in which case the code should not panic on error and cause revert.

Recommendation:

Revise behavior of the `submit_receipt` and `handle_interrupted_transfer` functions.

Status:

This issue has been resolved through https://github.com/gear-tech/gear-bridges/pull/270 .

Message Tracker / Message Statuses Redundancy

Project	Gear 2025: Q1: Gear Bridges
Type	PROTOCOL IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Likelihood	LOW
Status	ACKNOWLEDGED
Issue(s)	Message Tracker / Message Statuses Redundancy

Involved Artifacts:

- gear-programs/vft-manager/app/src/services/request_bridging/msg_tracker.rs
- gear-programs/vft-manager/app/src/services/request_bridging/token_operations.rs
- gear-programs/vft-manager/app/src/services/submit_receipt/msg_tracker.rs
- gear-programs/vft-manager/app/src/services/submit_receipt/token_operations.rs

Description:

The status tracking flow in both the `request_bridging` and `submit_receipt` methods uses similar code, even though the same action (mint/unlock) is needed after a failure in either case. This leads to unnecessary code duplication. Additionally, several statuses indicate the need for manual intervention (e.g., `TokenDepositCompleted(true)`, `BridgeResponseReceived(None)`, and `TokensReturnComplete(false)`), making them redundant.

Problem Scenarios:

Message Tracker Redundancy

MessageTracker code is duplicated for SubmitReceipt and RequestBridging. Differences are in

- statuses - `MessageStatus` enum, and
- sender/receiver field in `TxDetails`

the rest of the functionality is the same.

Statuses from SubmitReceipt `SendingMessageToWithdrawTokens` and `TokenWithdrawComplete` serve the same purpose as statuses `SendingMessageToReturnTokens` and `TokensReturnComplete` from RequestBridging, respectively. In a unified approach, `SendingMessageToReturnTokens` and `TokensReturnComplete` could be used for SubmitReceipt as well. If differentiation between the processes that created them is necessary, a new field could be added to the MessageInfo or the status value can be parameterized with operation: `SendingMessageToReturnTokens(Operation)`, `TokensReturnComplete(bool, Operation)`, where operation is a new enum with values `RequestBridging`, and `SubmitReceipt`.

Sender/Receiver field could be named as address.

Message Statuses Redundancy

Generally, our conclusion is that only final states are important and other states are not needed. Only from final states switch to the next final state is possible. The final states are `TokenDepositCompleted(bool)`, `BridgeResponseReceived(Option<U256>)` and `TokensReturnComplete(bool)`.

- `SendingMessageToDepositTokens` - this is a transitional state, this status is never saved.
- `TokenDepositCompleted(bool)` - this status is saved, and needed.
- `SendingMessageToBridgeBuiltin` - this is a transitional state, `TokenDepositCompleted(true)` can be used instead, as this was the last successful state.
- `BridgeResponseReceived(Option<U256>)` - if this state is reached, then it is saved. It means the process either deletes a message or continues with return, otherwise built-in send was unsuccessful and the last successful state is kept.
- `SendingMessageToReturnTokens` - this is a transitional state, the previous state can be kept. It is either of `TokenDepositCompleted(true)` or `BridgeResponseReceived(None)`.
- `TokensReturnComplete(bool)` - this state is saved, and needed.

Additionally, there are three dual states:

- `TokenDepositCompleted(bool)`, `BridgeResponseReceived(Option<U256>)`, and `TokensReturnComplete(bool)`

`false` and `None` can be mapped to a new state `ReturnTokens` and simplified states `TokenDepositCompleted`, `BridgeResponseReceived(U256)` and `TokensReturnCompleted` can be kept.

Recommendation:

Simplify the Message Tracker and consolidate message statuses.

No validation of config inputs

Project	Gear 2025: Q1: Gear Bridges
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Likelihood	LOW
Status	ACKNOWLEDGED
Issue(s)	No validation of config inputs

Involved Artifacts:

- gear-programs/vft-manager/app/src/services/mod.rs
- gear-programs/vft-manager/app/src/services/request_bridging/token_operations.rs

Description:

When Config is updated there is no embedded validation of its values. For example, zero values can be inserted.

Problem Scenarios:

A reply is not handled if zero is passed as a value of `reply_deposit` argument in send_bytes_with_gas_for_reply, which is used in request_bridging and submit_receipt.

Recommendation:

Validate all inputs to avoid an undesired behavior.

Findings Classification

Finding Categories

- *Protocol*: A flaw or problem in an abstract protocol or algorithm.
- *Implementation*: A problem with the source code. For example a bug, a divergence from a specification, or a poor choice of data structure.
- *Code structure:* A problem that impacts the extent to which the project is maintainable and understandable by developers in the long term.
- *Documentation:* A lack of documentation, or insufficient clarity, accuracy, understand-ability or readability of existing documentation.

Finding Categories

- *Informational*: The issue does not pose an immediate risk (it is subjective in nature). Findings with Informational severity are typically suggestions around best practices or readability.
- *Low*: The issue is objective in nature, but the security risk is relatively small or does not represent a security vulnerability.
- *Medium*: The issue is a security vulnerability that may not be directly exploitable or may require certain complex conditions in order to be exploited.
- *High*: The issue is an exploitable security vulnerability.
- *Critical:* The issue that disrupts functionality, security, or performance that leads to system failure, data loss, or security breaches.

Finding Severity Categories

Finding severity is derived from findings Impact and Likelihood as follows:

SEVERITY		LIKELIHOOD			
		NONE	LOW	MEDIUM	HIGH
IMPACT	NONE	INFORMATIONAL	INFORMATIONAL	INFORMATIONAL	INFORMATIONAL
	LOW	INFORMATIONAL	LOW	LOW	MEDIUM
	MEDIUM	INFORMATIONAL	LOW	MEDIUM	HIGH
	HIGH	INFORMATIONAL	MEDIUM	HIGH	CRITICAL

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Ethernal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Ethernal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bug free status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited. This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Ethernal to perform a security assessment.

This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.