Audit report

Gear 2024 Q3: Gear Bridges - ZK

Authors:

Andrija Mitrović

Sara Nikolić

Nikola Jovićević

Marko Jurić

Mile Kordić

Mladen Kovačević, PhD

Jelena Sekulić

August, 2024

Contents

Project Overview

Scope

In July and August 2024, Ethernal has conducted a security audit for Gear's implementation of zero knowledge prover and Ethereum smart contracts which are a part of a bridge between Vara and Ethereum. The audit aimed at inspecting the correctness and security properties of the solution. The components under scope are:

- zk prover (Vara to Ethereum)
- circuits
- ethereum smart contracts

The audit was performed from July 1st, 2024 to August 22th, 2024.

Relevant Code Commits

The audited code was from:

- commit hash d42251c3c9d94309a7855d6d774c6054a139a674.

Conclusion

After conducting a thorough review of the project, we found it to be carefully designed and generally well implemented. The audit was divided into three primary areas of focus: prover, circuits and Ethereum smart contracts. The whole audit process has been done through in advance agreed upon four phases:

1. System Design Analysis
2. Detailed Proof Analysis
3. Circuit Analysis
4. EVM Smart Contract analysis

The **System Design Analysis** phase focused on analyzing and decomposing the proofs generated by the relayer to uncover the relationships and connections between them. It also included an analysis of the Vara network to determine why specific proofs were necessary. The outcome of this phase was a comprehensive diagram illustrating the composition of the Final Proof and its constituent proofs, along with a detailed system overview of the prover functions.

The **Detailed Proof Analysis** phase evaluated whether the proof generation and composition identified in the previous phase were appropriate for generating the resulting proof. This phase involved an in-depth examination of individual proofs and their compositions, as well as the verification of all input data and constraints used by Plonky2 in circuit creation, without delving into the circuits themselves. The output included a thorough analysis of the coupling between prover functions, with a particular focus on any added constraints, and a report on any inconsistencies between the prover structures and functions.

The **Circuit Analysis** phase involved a cryptographic examination of the circuits used in proof generation. The analysis focused on the cryptographic libraries found in the circuits folder, with the aim of identifying any inconsistencies or vulnerabilities. The results of this phase were documented as a part of the report.

The **EVM Smart Contract Analysis** phase audited four EVM smart contracts responsible for processing Merkle roots along with accompanying zk proofs and transfers sent by users and relayers. The output consisted of a design and protocol analysis of these smart contracts, along with a report on any inconsistencies or vulnerabilities that were discovered.

Throughout the auditing process Ethernal team found in total 12 findings, of which 2 were HIGH in severity, 4 MEDIUM and the rest LOW and INFORMATIONAL.

Overall, it can be concluded that the project is of very high quality. Taking in to consideration the domain and that the scope was a part of the larger construction the code was well structured and easy to follow. The cooperation and expertise of the Gear's team greatly contributed to the success of the audit, and we believe the project is on a strong foundation moving forward.

Audit Dashboard

Target Summary

- Code and specification version:
  - Commit: d42251c3c9d94309a7855d6d774c6054a139a674
- Platform: Rust, Solidity

Engagement Summary

- Dates: 01.07.2024. to 22.08.2024.
- Method: Manual code review, protocol analysis
- Time Spent: 18 person weeks

Findings Severity Summary

Finding Severity	**#**
High-Severity Issues	2
Medium-Severity Issues	4
Low-Severity Issues	2
Informational-Severity Issues	4
Total	12

System Overview

Prover

Prover is a component intended to create two types of proofs:

- **Final Proof** - proof that message Merkle trie root is included in a `pallet-gear-bridge` storage at a specific finalized block;
- **LatestValidatorSet** - proof of validator set changes.

These two proofs, when combined, aim to enable trustless relaying of Merkle trie roots from `pallet-gear-bridge` storage to Ethereum.

The creation of proofs relies on proof composition, where proofs are calculated and validated before being used in higher level proofs. It also includes recursive proofs that use IVC (Incrementally Verifiable Computation), a concept that involves breaking the computation into smaller steps and proving them iteratively, such as in the case of tracking the validator set updates from genesis to the current block.

The Prover also uses the Circuit component, which defines different circuits for various purposes. More about this will be described later in the chapter.

The complexity of the Prover component and how proofs are aggregated is best illustrated in a Figure 1.

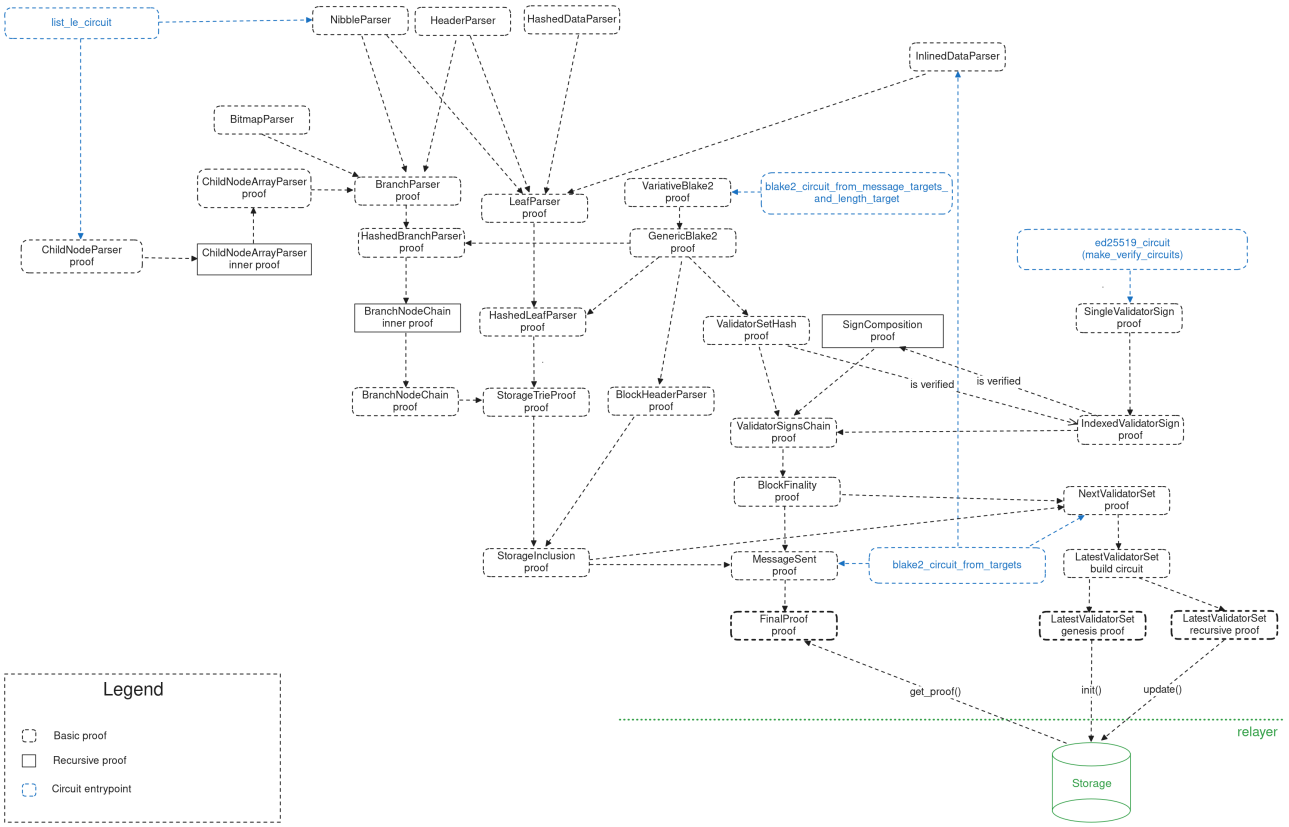

Figure 1. Proof Aggregation

This diagram can be divided into several logical units for easier explanation:

- Final proof and Message Sent Proof
- Latest Validator Set Proof
- Block Finality Proof
- Generic Blake2 Proof
- Storage Inclusion Proof
- Hashed Leaf Parser Proof
- Branch Node Chain Proof

Final Proof and Message Sent Proof

This presented part aims to establish a cryptographic proof that a specific message was correctly queued for relaying on Vara network, ensuring its presence in storage and the finality of the block it was included in. This proof can then be used for further verification or submission to Ethereum blockchain.

Final Proof combines the *Message Sent* proof with the *Latest Validator Set* proof (obtained from storage) to generate a final proof that ensures the message was queued for relaying. This final proof will be used for submission to Ethereum. This proof involves several steps and constraints to ensure its validity. Here are the constraints:

- The validator set and authority set IDs are consistent between the *Message Sent* proof and the *Latest Validator Set* proof.
- The genesis configuration values are correctly integrated and consistent with the *Latest Validator Set* proof.

Message Sent Proof combines proofs for block finality and message inclusion in storage, and provides a method to generate a proof that a message was present in a finalized block. This involves combining and verifying the storage inclusion proof and finality proof, and constructing the necessary public inputs.

- The block hash from the *Storage Inclusion* proof must match the block hash from the *Block Finality* proof.
- Ensures that the computed hash of the actual storage data matches the hash provided in the *Storage Inclusion* proof.

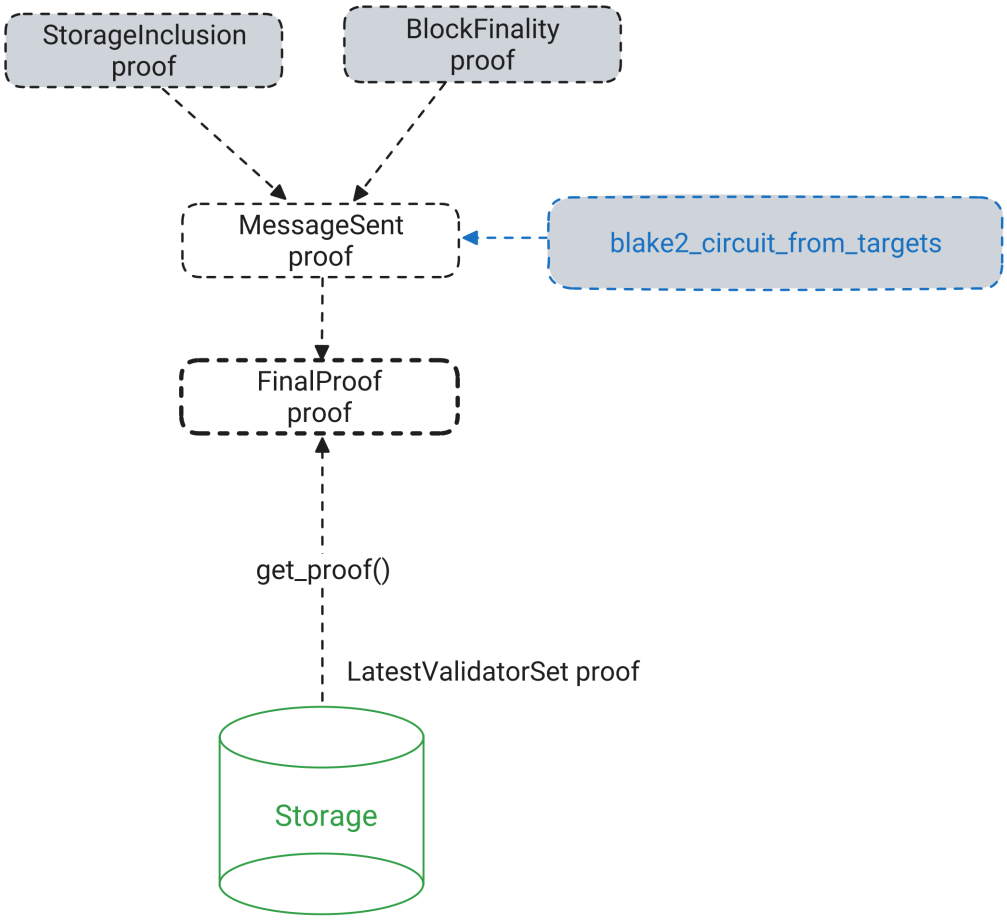

Figure 2. Final Proof and Message Sent Proof

Latest Validator Set Proof

Latest Validator Set is utilized to verify a sequence of validator set updates, following the transitions from the initial validator set to the current one. It is retrieved from storage and represents the input for the final proof. Additionally, it is used to verify if validators are correctly signing GRANDPA messages, used for block finalization.

Genesis proof proves the very first transition from genesis to its subsequent authority set. It is represented as a constant within the circuit and used as the initial value of the latest validator set proof in storage.

Recursive proof is used in the process of proving the validator set change. It generates the validator set change proof by using the proof of the transition of the validator set from the initial one to the current one and the proof of the inclusion of the next authority set into storage. This proof is later used in further steps of the recursion. The validator set change is proven within a loop where the composed proof of validator sets up to the current iteration is taken and passed further, generating the new composed proof.

Both `prove_genesis()` and `prove_recursive()` have to build the circuit first, against which they will create their proofs. This is done using the **build_circuit** function, which utilizes the *Next Validator Set* proof to construct a circuit that verifies validator set changes. It also defines several conditions that need to be satisfied so verification can pass:

- The genesis authority set id must be equal to the genesis set id defined in config;
- The genesis authority set hash must match the genesis hash, also defined in config;
- The current authority set id from the recursive proof, must match the one calculated under the *Next Validator Set* proof.

Next Validator Set proves that the validator set has changed. This change means that the current validator set finalized a block (*Block Finality* proof) containing the next validator set in the storage (*Storage Inclusion* proof). The circuit verifies that a majority of validators from the current set have set hash inclusion into the storage of `pallet-gear-bridge` and signed the vote for the change. It also includes several constraints that need to be satisfied:

- Block hash within the inclusion proof target must be the same as in the finality proof target's grandpa message;
- Hashed storage data target must be the same as `storage_item_hash` from inclusion proof target.

Next Validator Set uses `blake2_circuit_from_targets()` function from **plonky2_blake2b256** package in order to calculate the the hash for mentioned storage data target.

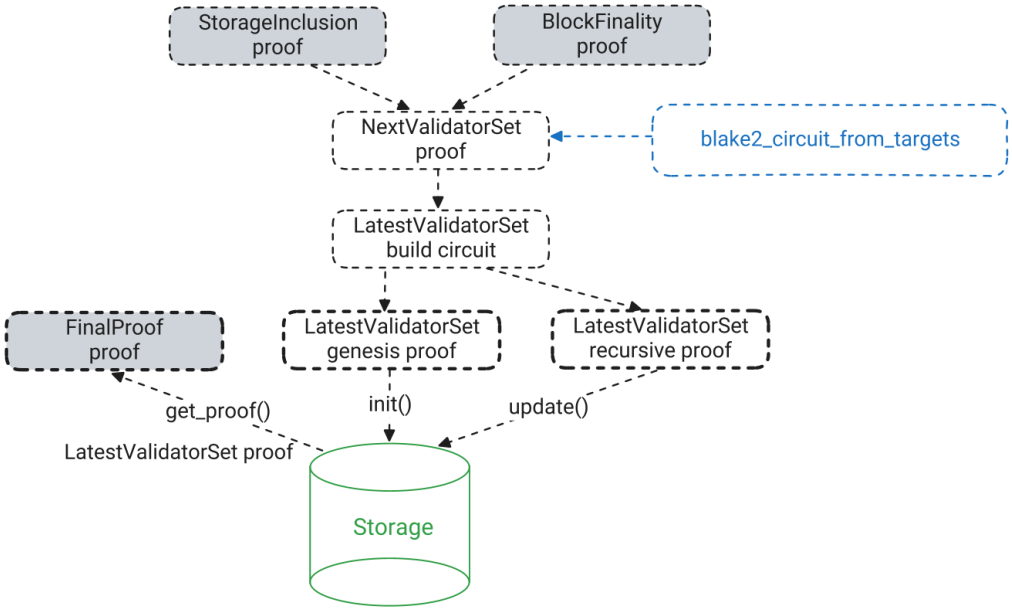

Figure 3. Latest Validator Set Proof

Block Finality Proof

Block Finality aims to prove that a specific block was finalized on the Gear blockchain by a validator set. This process involves confirmation that a majority of validators, more than 2/3 of them, have signed the GRANDPA vote for the block. For this purpose, the *Validator Signs Chain* proof is used. The *Block Finality* proof involves a constraint that requires the signed GRANDPA message to be in the pre-commit phase of voting.

Validator Signs Chain, as mentioned, proves that a majority of validators from the current validator set have signed the GRANDPA message. It further uses several circuits within the execution:

- **Validator Set Hash** proof, that is used to verify that the hashing of the validator set is correct. It uses the *Generic Blake2* proof to confirm that the Blake2 hash of generic-length data is correct. The only constraint that needs to be checked additionally is that the length of the proof calculated under *Generic Blake2* matches the desired one, which is the `number of validators in the set * 32 (ED25519_PUBLIC_KEY_SIZE)`.
- **Indexed Validator Sign** proof confirms that a validator with a particular index in the validator set has signed the GRANDPA message. For this purpose, it uses the **Single Validator Sign** proof, which proves that a single validator has signed the GRANDPA message. The public key of this validator is then compared to the public key of the validator at the specific index in the validator set, and they must match. Additionally, since proving the *Single Validator Sign* circuit is the most time-consuming proof among all the others, the circuit is built only on the first call to prove and retrieved from the cache for subsequent calls. *Indexed Validator Sign* is proven in parallel, for each validator that has signed the message, and those proofs are collected after the execution. They are then passed to the *Sign Composition* circuit.
- **Sign Composition** proof, which is a recursive proof, is built upon the *Indexed Validator Sign* proofs. Each proof passed into its build function is verified, and its message, validator set hash, and validator count must match the ones from the public inputs. Additionally, during each iteration, it will increment the number of verified signatures and include it in the final *Sign Composition* target, which will be checked against another public input.

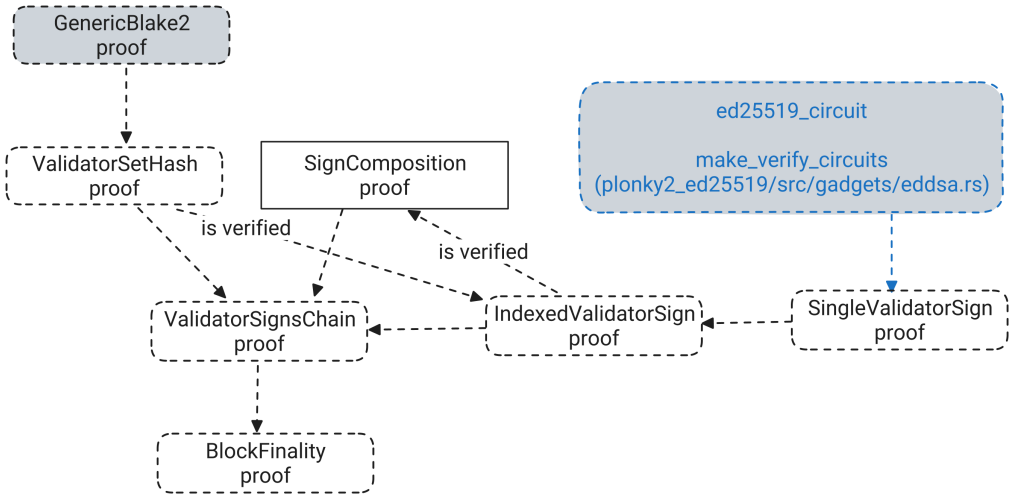

Figure 4. Block Finality Proof

Generic Blake2

Generic Blake2 circuit is used to compute Blake2b-256 hash of generic-length data. Blake2b-256 algorithm splits message data into data blocks while calculating hash, where each block has 128 bytes. Therefore, *Generic Blake2* calculates number of blocks in message data and checks that it's less than maximum block count.

Afterwards, the algorithm generates *Variative Blake2* inner proof, and then verifies it. Finally, public inputs for the circuit are created using public inputs from *Variative Blake2* proof.

Variative Blake2 computes Blake2 hash of variative-length data. First, it pads message data with zeroes to fit in desired length (MAX DATA BYTES = 1024) and converts it into Byte Targets. Then, it creates Blake2 circuit to calculate Blake2 hash of padded message data.
`blake2_circuit_from_message_targets_and_length_target` function from *plonky2_blake2b256* library is used for that purpose. Finally, public inputs for *Variative Blake2* circuit are created:

- *data* - message data transformed to padded Byte Targets,
- *length* - non-padded message data length,
- *hash* - Blake2 hash of message data.

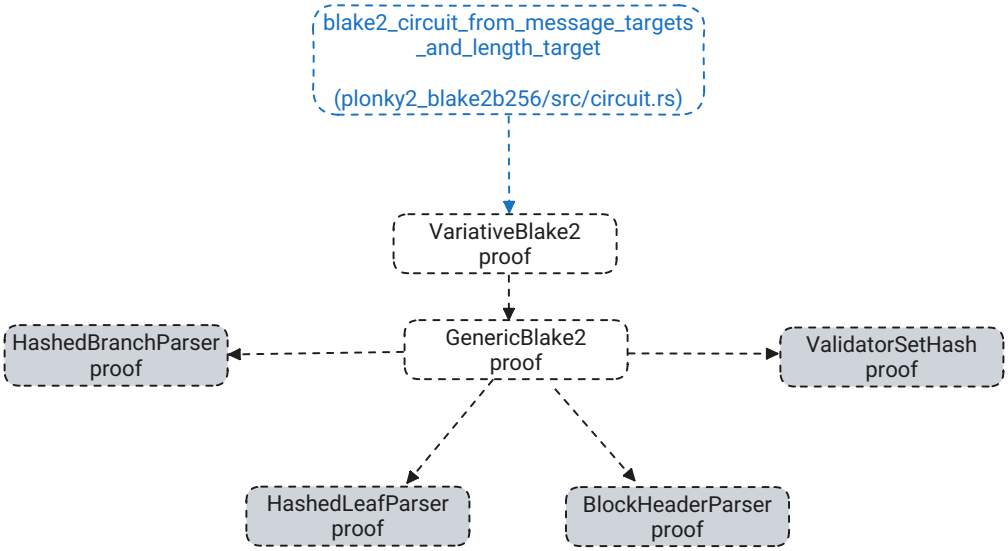

Figure 5. Generic Blake2

Storage Inclusion Proof

The overall purpose of presented part is to construct zero-knowledge proofs for verifying the inclusion of specific data within a storage trie and ensuring the integrity of block headers. The process involves parsing and hashing various components, constructing recursive proofs, and finally composing these proofs.

Storage Inclusion proves that a specific data item is included in the storage trie. It combines proofs from both the *Block Header Parser* and the *Storage Trie Proof* to ensure the integrity and inclusion of data within the storage trie. Each node in the trie is indexed by a sequence of nibbles (4 bits, or half of a byte), allowing for efficient storage and retrieval of data. This proof involves several constraints to ensure its validity:

- The state root from the block header must match the root hash of the storage trie. This constraint ensures that the storage trie is correctly tied to the block header by making sure the state root in the block header matches the root hash of the storage trie.
- The provided storage address must match the address derived in the storage trie proof. This constraint ensures that the address used to access the storage item in the trie is correctly verified against the provided address nibbles.

Block Header Parser extracts the state root from an encoded block header and verifies the block hash. It creates a proof for the block header using *Generic Blake2* proof.

Storage Trie Proof represents a proof that an item is included in a storage trie. It composes proofs for branch nodes and leaf nodes into a single proof using recursive verification. These constraints are used to verify that the proofs are consistent with each other and with the expected outcomes:

- Ensures that the leaf hash from the branch node chain matches the node hash from the hashed leaf parser.
- Ensures that the partial address derived from the branch node chain matches the partial address used in the hashed leaf parser.

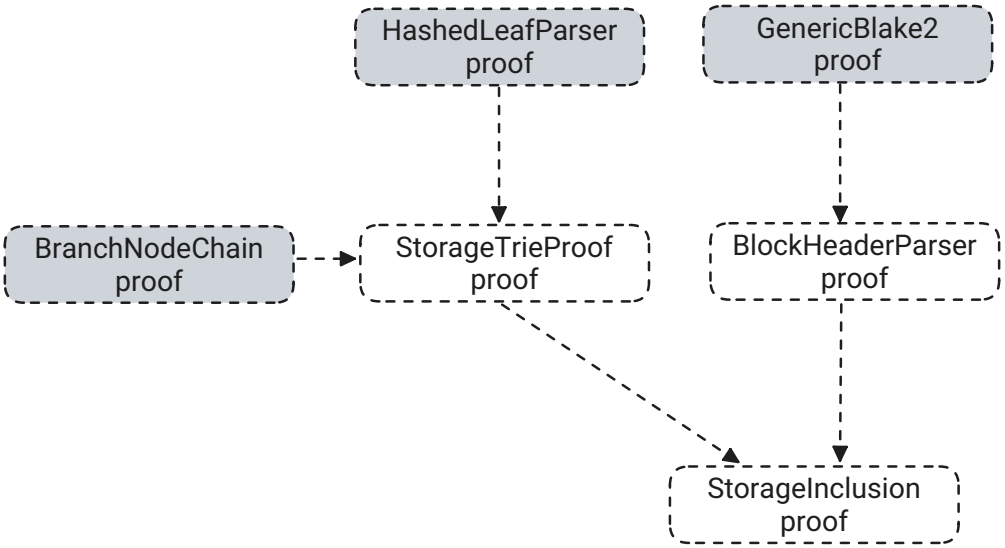

Figure 6. Storage Inclusion Proof

Hashed Leaf Parser Proof

Presented part defines a circuit to prove the correct parsing of a leaf node in a Merkle tree. It uses the plonky2 library to construct and verify cryptographic proofs.

Hashed Leaf Parser Proof generates a proof for the node data hash using a *Generic Blake2* hasher. Then it combines this with the *Leaf Parser Proof*. Here are the constraints involved:

- The length of the node data used in the *Leaf Parser* proof and the hasher proof must be the same. This ensures that both proofs are working with the same amount of data
- The actual data used in the Leaf Parser proof and the hasher proof must be identical. This involves iterating through the node data from both proofs and connecting corresponding elements.

Leaf Parser Proof processes the node data and generates a proof to confirm the correctness of this parsing. Four parsers are used for this purpose:

- Header Parser,
- Nibble Parser,
- Hashed Data Parser and
- Inlined Data Parser.

These parsers will be discussed in detail in the following sub-chapters.

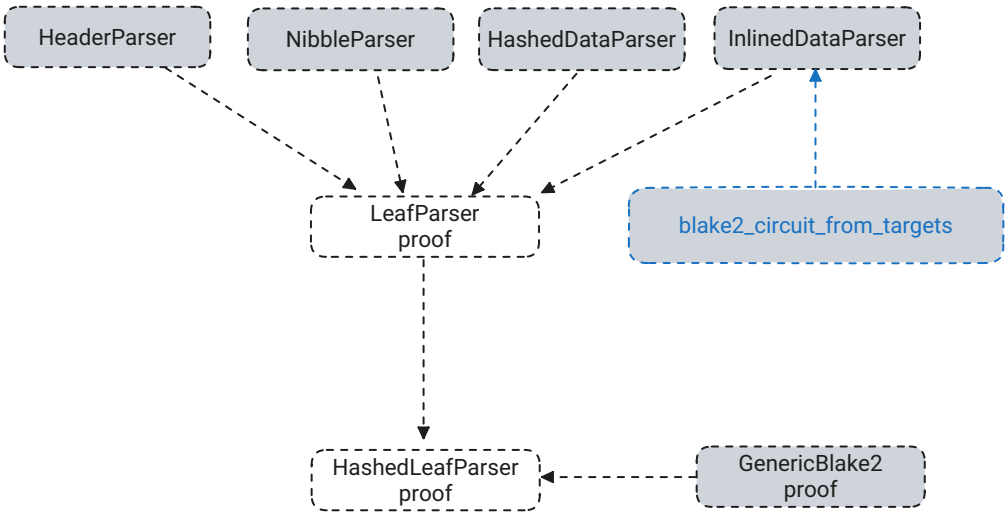

Figure 7. Hashed Leaf Parser Proof

Branch Node Chain Proof

Branch Node Chain proof is used to prove the correct parsing of branch nodes on the path from the root node to a leaf node. Leaf nodes contain hashed values of data stored in pallet-gear-bridge storage, whereas branch nodes do not contain any value but help navigate the structure of the Merkle trie.

Branch Node Chain proof is created recursively for each branch node on the path using IVC, e.g. for every branch node an inner *Hashed Branch Parser* proof is created and proved recursively. Furthermore, addresses of branch nodes on the path are concatenated to form Partial Storage Address, which, alongside with root node hash and Blake2 hash of encoded leaf data, compose public inputs for Branch Node Chain proof.

Branch Node Chain circuit forces the following constraints:

- Node hash of an inner *Hashed Branch Parser* proof must be equal to the *leaf hash* public input of composed Branch Node Chain proof in current iteration of IVC.
- Partial address of an inner *Hashed Branch Parser* proof must be equal to *partial address* public input of composed Branch Node Chain proof in current iteration of IVC.
- public inputs of composed Branch Node Chain proof for the next iteration of IVC are defined:
    - *root hash* is equal to the root hash in the current iteration,
    - *leaf hash* is equal to the hash of node that's been processed in the current iteration,
    - *partial address* is equal to the concatenated addresses of nodes from root to node in current iteration.

Hashed Branch Parser proof is used to prove correct parsing of a single branch node. First, *GenericBlake2* circuit is used to calculate Blake2 hash of branch node data. Second, non-hashed branch parser proof is generated with *padded node data* public input, which represents encoded node data. Finally, a constraint is set that the *branch node data* from *Generic Blake2* circuit must be equal to *padded node data*.

Branch Parser is used to parse branch node data. First, it parses node metadata, which contains data about node's children and it's used in *Child Node Array Parser*, which parses children of the branch node. Afterwards, *Header Parser* takes first two bytes from branch node data to parse Node Header, which is then used by *Nibble Parser* to parse nibbles that form Partial Storage Address alongside with nibbles from the other branch nodes on the path from root to leaf. Finally, *Bitmap Parser* parses bitmap that describes nibbles of child nodes.

Branch Parser also forces several constraints:

- Node data target must have the same value as branch node data.
- Children nodes' data must be placed immediately after bitmap data in byte array of branch node data.
- There mustn't be any additional bytes after children nodes' data in byte array of branch node data.
- Bitmap Parser calculates the same number of children nodes as Child Node Array Parser.
- Bitmap Parser calculates the same index of the next child node to process in children nodes array as Child Node Array Parser.
- Child node hash target must have same value as Child Node Array Parser's public input `claimed_child_hash`.

Child Node Array Parser parses children nodes of the branch node that is being processed in the current IVC iteration. This is also done recursively for each child node using IVC, e.g. for each child node an inner *Child Node Parser* proof is created. Additionally, overall number on children nodes is calculated. While iterating through children nodes array, Child Node Array Parser finds the next child node to be processed by *Branch Parser* and sets its index as a public input.

Child Node Parser is used to parse a single child node. It reads child node data and check if it's the node that should be processed in the next iteration of IVC. If that's the case, it checks the length of child node encoded data and checks that its hash matches the *claimed child hash* target in Child Node Parser circuit.

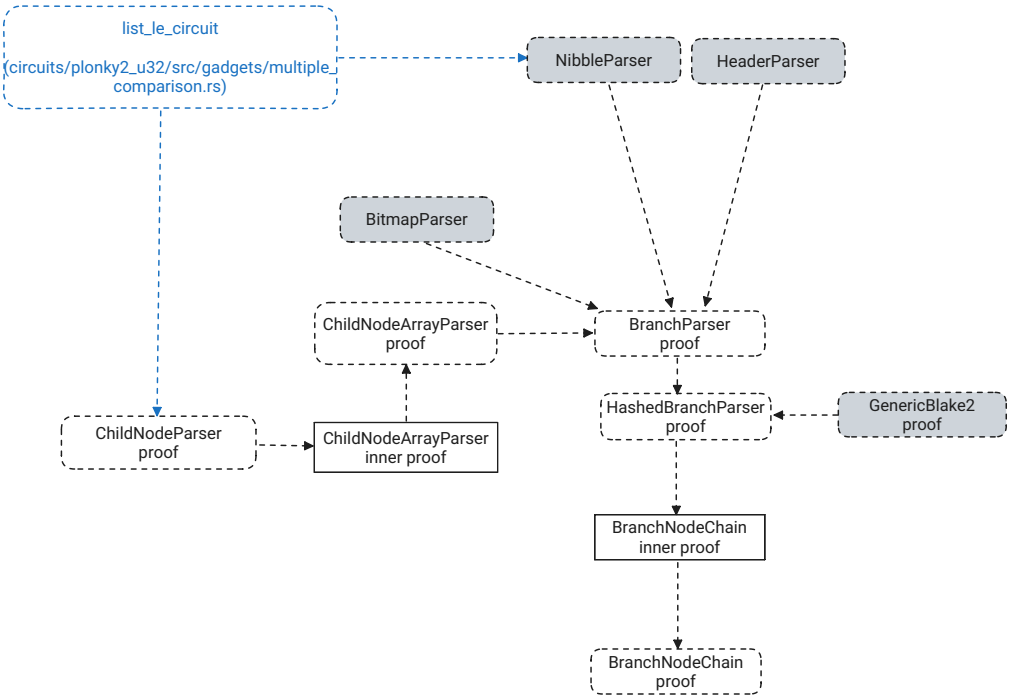

Figure 8. Branch Node Chain Proof

Parsers

There are 5 parsers used to parse data from Merke trie branch and leaf nodes:

- Header Parser,
- Nibble Parser,
- Hashed Data Parser,
- Inlined Data Parser and
- Bitmap Parser.

Every parser tracks the amount of bytes that was read from node data while processing it, so that the next parser knows the offset that the next bytes of node data should be read from.

Header Parser

Header Parser is used to parse node prefix (header), which is guaranteed to be located in the first 2 bytes of node data, for both *branch* and *leaf* nodes (note that it can be 1 or 2 bytes long). First 2 or 3 bits of prefix define node type - *BranchWithoutValue*, *HashedValueLeaf* and *Leaf*. The rest of the prefix contains number of nibbles that Nibble Parser should read in order to parse partial address from node data.

Nibble Parser

Nibbles are half-bytes that hold the information about storage address. Although each nibble is stored as a byte (*u8*), they can have values from 0 to 15. Storage address is composed of address nibbles written in nodes on the path from root to leaf in a Merkle trie. Number of nibbles that Nibble Parser should read and parse from node data is written in node header and parsed by Header Parser.

Hashed Data Parser

Hashed Data Parser parses leaf nodes' hashed values of the stored data. This parser is used to parse data from only leaf nodes whose type is *Hashed Value Leaf*. Data hash is expected to have 32 bytes length and it's parsed to Blake2 target. The rest of the node data is padding, since the whole node data is expected to have 128 bytes.

Inlined Data Parser

Inlined Data Parser is used to parse storage data from nodes whose type is *Leaf*. First byte that Inline Data Parser reads from node data contains the information about storage data length, which is expected to be 32 bytes. After reading 32 bytes of storage data from leaf node, Inlined Data Parser uses them to create Blake2 hash.

Bitmap Parser

Bitmap is a 2-byte data in a branch node that describes nibbles of child nodes. Nibbles represent positions of bits with the value of 1 within the bitmap. For example, if bitmap value is `0b00_11_10_00_11_00_01_10` then it means that only child nodes with nibbles 2, 3, 4, 8, 9, 13, 14 exist. Bitmap Parser computes overall amount of children nodes as number of ones in the bitmap. Additionally, it uses nibble of the next child node in traversal of the Merkle trie to find the corresponding index in the array of children nodes (every branch node has an array of its children nodes). For example, if we have bitmap `0b00_11_10_00_11_00_01_10` and provide `nibble = 4` as input, we should get `index = 2` as empty (non-existent) nodes aren't stored in this array.

Circuits

Circuits component consists of five separate packages:

- plonky2_blake2b256
- plonky2_ecdsa
- plonky2_ed25519
- plonky2_sha512
- plonky2_u32

All circuits are built by using plonky2 library. Plonky2 is a SNARK implementation based on techniques from PLONK and FRI. It is the successor of Plonky, which was based on PLONK and Halo. Plonky2 is built for speed, and features a highly efficient recursive circuit. The main structure from plonky2 that is used for building circuits (provides necessary tool kits that enable one to design circuits with its associated prover/verifier data) is `CircuitBuilder`. In the following text we provide descriptions of the aforementioned components.

plonky2_blake2b256

The main objective of this project is to implement the BLAKE2b-256 hash function as a circuit. BLAKE2 is an improved version of the SHA-3 finalist BLAKE optimized for speed in software. It is often faster than MD5, yet provides security similar to that of SHA-3. BLAKE2b is optimized for 64-bit platforms and BLAKE2b-256 produces message digests of length 256 bits.

Circuit is built in the `blake2_circuit_from_message_targets_and_length_target` function. This function uses the message along with its length, encoded to bytes, to create the message digest, i.e. hash. It is used for the creation of several proofs within the **Prover** component, such as `VariativeBlake2` and `MessageSent`, as well as for `define()` function of `inlined_data_parser`.

plonky2_ecdsa

Functions from *gadgets/biguint.rs* are referenced from the **plonky2_ed25519** component, being the only code from this project actually used. Biguint gadget provides necessary functions for arithmetic operations on big unsigned integers, used for base and scalar field arithmetic for ed25519 curve.

plonky2_ed25519

This project contains the implementation of the Ed25519 digital signature algorithm, which is an instance of the EdDSA signature scheme, that uses Twisted-Edwards curve ed25519 and SHA-512 as a hash function. It is designed to be faster than existing digital signature schemes without sacrificing security.

This project contains a regular implementation of the mentioned curve, with point addition, doubling and scalar multiplication, as well as all of that implemented for proof generation (as a circuit). Main function is `make_verify_circuit`, which is called from the `SingleValidatorSign` build function. This function builds a circuit for verification of Ed25519 signatures. Specifically, it is used to verify signature of GRANDPA message for each validator in current validator set. It takes message length as a parameter, and returns the set of targets - message, signature and public key targets, encapsulated in `EDDSATargets` structure. Hashing is also done inside a circuit, concretely SHA-512 circuit, described in the next paragraph.

plonky2_sha512

In this project, SHA-512 circuit is designed. SHA-512 is one of the hash functions from SHA-2 family. It produces a hash of length 512 bits.

The name of the main function is `sha512_circuit`. It takes message length in bits as input and outputs `Sha512Targets` structure, which consists of message and digest targets. This circuit is used as a part of Ed25519 signature verification circuit.

plonky2_u32

This project can be considered as a plonky2 extension, since it contains several additional gate types, used for the implementation of basic arithmetic operations on unsigned 32-bit integers. Gadgets from this project are used in: **plonky2_blake2b256**, **plonky2_sha512**, **plonky2_ecdsa** and **plonky2_ed25519**, as well as in **StorageTrieProof**.

Ethereum Smart Contracts

Ethereum smart contracts are designed to process messages and their corresponding proofs relayed from Gear-based blockchains. On the Gear-based blockchain, each message is stored in a binary Merkle trie. The Merkle root of this trie, which serves as a compact and verifiable summary of all messages, is then recorded in the `pallet-gear-bridge` storage. This setup ensures that messages can be reliably verified and processed when they are relayed to Ethereum.

The *Prover* component generates a zero-knowledge proof (ZK-proof) confirming the inclusion of the Merkle trie root in the storage at a specific finalized block. This proof is then submitted to the Ethereum. There are five Ethereum smart contracts:

- Relayer,
- Verifier,
- MessageQueue,
- ERC20Treasury and
- ProxyContract.

The `Relayer` contract handles the proof of Merkle trie root inclusion by forwarding it to the `Verifier` smart contract, which uses the `PlonkVerifier` for verification. Upon validating the proof, the `Relayer` then stores in memory the Merkle trie root and the number of the block containing the message.

Once the proof is verified and the Merkle trie root is stored in the `Relayer` contract, users can request to relay a message further onto Ethereum. To do this, they must provide proof of message inclusion in the Merkle trie to the `MessageQueue` contract. The `MessageQueue` contract ensures each message is processed only once and verifies the message by computing the Merkle root. It then compares the computed Merkle root with the one stored in the `Relayer` contract. If they match, the `MessageQueue` contract invokes the `IMessageQueueReceiver` contract to process the message, using the receiver address specified in the message.

The `ERC20Treasury` smart contract implements the `IMessageQueueReceiver` interface and serves as a vault for ERC-20 tokens, allowing users to deposit tokens and handle withdrawal by processing Vara message sent by the `MessageQueue` contract. It ensures that only authorized messages with a valid format are processed.

The `ProxyContract` is an upgradeable proxy implementation based on OpenZeppelin's ERC1967Proxy, which enables upgrading the implementation of a contract and calling functions on the new implementation.

Threat Inspection

Circuit Invariant Analysis

Invariant 1. Blake2b-256 implementation follows the specification and reference code given in RFC 7693.

Conclusion: Invariant holds. Blake2b-256 is implemented according to the specification and reference code given in RFC 7693. All parameters, constants and variables match the specification. Helper functions (F, G) as well as the main function match the specification. Plonky2 CircuitBuilder is properly used for circuit design.

Invariant 2. SHA512 implementation follows the specification and reference code given in RFC 6234.

Conclusion: Invariant holds. SHA512 is implemented according to the specification and reference code given in RFC 6234. All parameters, constants and variables match the specification. Helper functions (CH, MAJ, BSIG0, BSIG1, SSIG0 and SSIG1) as well as the main function match the specification. Plonky2 CircuitBuilder is properly used for circuit design.

Invariant 3. Twisted Edwards curves and edwards25519 curve implementation follows the specification given in Twisted Edwards Curves paper and RFC 7748.

Conclusion: Invariant does not hold. Twisted Edwards curves and edwards25519 curve are not implemented according to the specification given in Twisted Edwards Curves paper and RFC 7748. We checked the Twisted Edwards curves definition, edwards25519 curve definition, base and scalar field definitions of the edwards25519 curve, addition, doubling and scalar multiplication formulas. Issues related to this invariant are detailed in the following findings:

- *Issues Related to Twisted Edwards Curve Implementation;*
- *Issues Related to Base and Scalar Field Implementations;*
- *Missing Range-Checks.*

Invariant 4. EdDSA implementation follows the specification and reference code given in High-speed high-security signatures paper and RFC 8032.

Conclusion: Invariant does not hold. Verification of EdDSA signature is implemented according to High-speed high-security signatures paper and RFC 8032. Plonky2 CircuitBuilder is properly used for circuit design. However, EdDSA is affected by the underlying curve implementation issues mentioned in the previous invariant analysis. Furthermore, two issues were found in the EdDSA circuit itself. Those are further described in the following findings:

- *Missing Scalar Reduction in EdDSA Implementation;*
- *Point Decompression Can Cause the Code to Panic.*

Invariant 5. u32 library contains implementation of unsigned 32-bit integer arithmetic functions that is in line with the plonky2 library.

Conclusion: Invariant holds. All existing gates are properly defined and used in the corresponding gadgets.

Invariant 6. Biguint gadget contains implementation of big unsigned integer arithmetic functions that is in line with the plonky2 library.

Conclusion: Invariant does not hold. Implementation of big unsigned integer arithmetic is in line with the plonky2 library in all the functions except one, which we reported within the finding:

- *Missing Range-Checks.*

Prover Invariant/Threat Analysis

Invariant 1. Data parsing is done in accordance with decode_plan function from polkadot-sdk.

Threat 1: The Header Parser doesn't parse the *node type* in accordance with decode_plan function from polkadot-sdk.

Conclusion: This threat does not hold. The Header Parser defines a constraint that verifies whether the node type matches the prefix extracted from the node data header. This prefix comprises 2 or 3 bits from the first byte of node data.

Threat 2: The Header Parser doesn't parse the *nibble count* in accordance with decode_plan function from polkadot-sdk, causing the Nibble Parser to receive invalid inputs.

Consequences: The Nibble Parser may receive a *nibble count* input that exceeds the maximum allowable value.

Conclusion: This threat holds. The remainder of the first byte, after the prefix, is allocated for the nibble count, potentially extending into the second byte, which is enough for storing nibble count that is greater than `MAX_STORAGE_ADDRESS_LENGTH_IN_NIBBLES = 64`. Further details on this issue are provided in the following finding:

- *Nibble Count in Nibble Parser May Exceed the Maximum Allowable Value*

Threat 3: The Nibble Parser doesn't parse the *partial address* in accordance with decode_plan function from polkadot-sdk. Specifically, it lacks a constraint to ensure that the first nibble is set to zero when representing padding.

Consequences: The Nibble Parser fails to detect node data with an incorrect format.

Conclusion: This threat holds. The Nibble Parser doesn't contain a constraint to ensure that nibble padding is set to zero when the nibble count is an odd number. Further details on this issue are provided in the following finding:

- *Nibble Padding Should Be Set to Zero.*

Threat 4: The Bitmap Parser doesn't parse the *bitmap* in accordance with decode_plan function from polkadot-sdk.

Conclusion: This threat does not hold. The bitmap comprises 2 bytes. Each byte is encoded so that its bits should be read from least to most significant in order to correctly decode it and determine the position of the node's children in the children array. For example, if bitmap value is `0b00_11_10_00_11_00_01_10` then it means that only child nodes with nibbles 2, 3, 4, 8, 9, 13, 14 exist. In this case, the first byte in the bitmap is `0b00_11_10_00`, and its encoded value is `0b00_01_11_00` (bits sorted from least to most significant). Also, the second byte in bitmap is `0b11_00_01_10`, and its encoded value is `0b01_10_00_11`. A small program has been implemented to verify this claim. It encodes the bitmap value and then prints the bits from the first and second byte of the encoded bitmap.

Threat 5: The Hashed Data Parser doesn't parse the *hash data* in accordance with decode_plan function from polkadot-sdk.

Conclusion: This threat does not hold. The Hashed Data Parser reads a *Blake2* hash value and expects it to be 32 bytes (`BLAKE2_DIGEST_SIZE`) long.

Threat 6: The Inlined Data Parser doesn't parse the *data* in accordance with decode_plan function from polkadot-sdk.

Conclusion: This threat does not hold. In-lined data length parsed from the first byte must be equal to `INLINED_DATA_LENGTH` (32 bytes). The data hash is generated using the `plonky2_blake2b256` circuit.

Invariant 2. Only trusted data is the genesis data that is sourced from config. Genesis data must be the same on the relayer and the prover.

Threat 1: A Byzantine actor could use an incorrect `GENESIS_CONFIG` input to submit a Byzantine proof to Ethereum smart contracts.

Conclusion: This threat does not hold. Authority Set ID and Validator Set Hash from Genesis Config are defined as **constants** while building the Final Proof circuit. Changing Genesis Config modifies the resulting circuit digest. The verifier checks the circuit digest against the expected digest to confirm that the proof was generated for the correct circuit. This means that if the prover generates a proof using an incorrect Genesis Config, the resulting circuit digest will not match the digest expected by the verifier.

Invariant 3. More than 2/3 of validators have signed GRANPA message for finalization of a block.

Threat 1: Block can be finalized with less than 2/3 of validators signing the GRANDPA message.

Conclusion: This threat does not hold. Need for more than 2/3 of validators is enforced here.

Threat 2: Block is not finalized even if more than 2/3 of validators have signed GRANDPA message.

Consequences: The prover could end up rejecting valid messages.

Conclusion: This threat holds. The validation process relies on sampling 2/3 + 1 of the validators from the set to validate message signatures, which is the minimum number required for the message to be accepted. However this means that if at least one of the sampled validators did not sign the message correctly (i.e., no proof can be generated to confirm their signature on the GRANDPA message), the proof generation process will halt. This occurs even if there are enough signatures from other validators from the set to provide the necessary votes for acceptance. Further details on this issue are provided in the following finding.

- *A Block with Enough Validator Votes on GRANDPA Message Can Be Rejected as Finalized.*

Invariant 4. All prover functions must have integrity (i.e. they to not trust input data, input data is verified before use).

Threat 1: Prover input `message_contents` can't be interpreted as a Blake2 hash.

Conclusion: The threat does not hold. `message_contents` is a byte vector representing the Blake2 hash of the root of the Merkle trie stored in the `pallet-gear-bridge` storage. Therefore, the `message_contents` vector must contain exactly `BLAKE2_DIGEST_SIZE` bytes, which is 32 bytes. This is ensured by defining a constraint that forces `message_contents` to be equal to `storage_item_hash` public input from `StorageInclusion` proof. The `storage_item_hash` is extracted from a leaf node data by either the Hashed Data Parser or the Inlined Data Parser. Both parsers ensure that the extracted hash is exactly 32 bytes in length.

Threat 2: Prover input `next_validator_set_data` can't be interpreted as a Blake2 hash.

Conclusion: The threat does not hold. `next_validator_set_data` is a byte vector representing the Blake2 hash of the next validator set stored in the `pallet-gear-bridge` storage. Therefore, the `next_validator_set_data` vector must contain exactly `BLAKE2_DIGEST_SIZE` bytes, which is 32 bytes. This is ensured by defining a constraint that forces `next_validator_set_data` to be equal to `storage_item_hash` public input from `StorageInclusion` proof. The `storage_item_hash` is extracted from leaf node data by either the Hashed Data Parser or the Inlined Data Parser. Both parsers ensure that the extracted hash is exactly 32 bytes in length. Furthermore, there is a constraint that the validator set hash of the current IVC (Incremental Verifiable Chain) iteration must match the next validator

set from the previous IVC iteration when proving the validator set change from genesis to the current validator set.

Threat 3: Prover input `block_finality_proof` contains empty `validator_set` or `pre_commits` vector.

Conclusion: The threat does not hold. There is a constraint that the Blake2 hash of the validator set must match the `current_hash` value from the Latest Validator Set proof. This constraint ensures that the validator set used in the Final proof is consistent with the validator set provided by the Latest Validator Set proof. Furthermore, the `pre_commits` vector cannot be empty, as its length must exceed 2/3 of the `validator_set` length. Even if it were possible that the `validator_set` is an empty vector, the formula defining the minimum number of pre-commits dictates that `pre_commits` should contain at least one element when the length of the `validator_set` vector is zero.

Threat 4: Prover input `block_finality_proof` contains invalid message

Conclusion: This threat does not hold because there is a constraint that verifies the block hash of the message signed by the GRANDPA voters against the one in the storage inclusion proof. If these two match, it ensures that the message to be executed is already part of the block. Additionally, the authority set ID received from the relayer is checked against the expected value (previous authority set ID + 1).

Invariant 5. Error handling in the prover component is robust and effective.

Threat 1: Parallel generation of *ValidatorSignsChain* proofs may lead to errors and inconsistencies.

Consequences: Race conditions and errors among threads executing in parallel could cause inconsistencies and errors.

Conclusion: This threat holds. Errors and panics within the threads could halt the proof generation process because they are not properly handled. More details about this issue can be found in the following finding:

- *Inadequate Error Handling May Cause Outages and Halt the Proof Generation Process.*

Threat 2: Error handling ensures that all potential errors are properly managed without causing panics.

Consequences: If an error occurs during proof generation, the use of `unwrap()` could cause a panic, leading to unexpected crashes. This could halt the entire proof generation process, preventing the prover component from functioning properly.

Conclusion: This threat exists because the current implementation relies on the `unwrap()` function without proper error handling. Errors should be managed more gracefully to prevent a single point of failure from impacting the entire system. More about this is described under the finding:

- *Inadequate Error Handling May Cause Outages and Halt the Proof Generation Process.*

Ethereum Smart Contracts Invariant/Threat Analysis

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

Threats:

Threat 1: Proofs that do not represent a valid Vara state are accepted.

Threat 2: The verifier rejects valid proofs.

Conclusion: These threats do not hold. The verifier implementation has been reviewed against the PlonK paper, and the code correctly follows the non-interactive version of the protocol using Fiat-Shamir. However, we note the following:

1. The Solidity code is automatically generated from gnark, a zk-SNARK Go library. The gnark library initially implemented an older version of PLONK, which itself is not a security concern. Still, the code is undergoing the transition to the latest version and is actively being changed. Thus, make sure to regularly update the verifier as the gnark library evolves.
2. Each verifier contract contains a subset of the Structured Reference String, as well as cryptographic parameters and a verifying key that encodes the circuit. These are assumed to be specified safely and correctly. The circuit itself is also out of the scope of this audit and is assumed to function as specified.

Invariant 2. Ethereum Relayer smart contract is not prone to replay attacks, i.e. can not accept a different merkle root and block number if these were already verified and stored.

Threat: Ethereum Relayer smart contract can accept different Merkle root and block number if these were already verified and stored.

Conclusion: This threat does not hold. Different/invalid combination of Merkle root, block number and ZK proof will not pass the ZK proof verification. Only a valid combination will pass, but the same values will be stored in the `_block_numbers` and `_merkle_roots` mappings. What has been noted is that the call to the gnark verifier can be skipped if checking that the Merkle root and block number have already been processed. More details in the following finding:

- *Relayer Smart Contract Unnecessarily Verifies Already Verified and Stored Merkle Roots with Their ZK Proofs.*

Invariant 3. Ethereum Message Queue smart contract rejects replay attacks.

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

Conclusion: This threat does not hold. Each Vara message whose proof has been successfully verified is marked as processed and sent to `ERC20Treasury` smart contract for further processing. Attempting to process a message that has already been marked as processed will revert with a `MessageAlreadyProcessed` error.

Invariant 4. Ethereum smart contract processes only messages that are verified against a trusted Merkle root.

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

Conclusion: This threat does not hold. First, the `MessageQueue` smart contract fetches the trusted Merkle root from the `Relayer` smart contract. Then it calculates the Merkle root using the provided Merkle tree proof for the Vara message and compares it to the trusted Merkle root, ensuring that only messages with a valid Merkle proof of inclusion are processed.

Invariant 5. Ethereum smart contract is resilient to invalid proofs, i.e. invalid proofs don't block further processing of valid ones.

Threat: A Vara message 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. Attempting to process a Vara message that has already been processed will result in a revert. However, a message is marked as processed only after its proof has been successfully verified.

Findings

Title	Type	Severity	Status
Missing Scalar Reduction in EdDSA Implementation	IMPLEMENTATION	HIGH	RESOLVED
Nibble Count in Nibble Parser May Exceed the Maximum Allowable Value	IMPLEMENTATION	HIGH	RESOLVED
Inadequate Error Handling May Cause Outages and Halt the Proof Generation Process	IMPLEMENTATION	MEDIUM	ACKNOWLEDGED
Point Decompression Can Cause the Code to Panic	IMPLEMENTATION	MEDIUM	ACKNOWLEDGED
A Block with Enough Validator Votes on GRANDPA Message Can Be Rejected as Finalized	IMPLEMENTATION	MEDIUM	ACKNOWLEDGED
Nibble Padding Should Be Set to Zero	IMPLEMENTATION	MEDIUM	RESOLVED
Missing Range-Checks	IMPLEMENTATION	LOW	RESOLVED
Issues Related to Base and Scalar Field Implementations	IMPLEMENTATION	LOW	RESOLVED
Issues Related to Twisted Edwards Curve Implementation	IMPLEMENTATION	INFORMATIONAL	RESOLVED
Relayer Smart Contract Unnecessarily Verifies Already Verified and Stored Merkle Roots with Their ZK Proofs	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED
Usage of Unaudited and Partially Audited Libraries	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED
Code Improvements	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED

Missing Scalar Reduction in EdDSA Implementation

Project	Gear Bridges - ZK - Circuits
Type	IMPLEMENTATION
Severity	HIGH
Impact	HIGH
Exploitability	HIGH
Status	RESOLVED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/5

Involved Artifacts:

- plonky2_ed25519/src/gadgets/eddsa.rs

Description:

The digital signature system, which utilizes Twisted Edwards curves, was checked against the methodology introduced in the paper. Additionally, RFC 8032 was used as a reference as well.

We have noticed the following inconsistency:

- Following the verification part of the algorithm, we noticed that the scalar s is missing reduction modulo scalar field order. This can be done like here.

Problem Scenario:

As explained in the Description section.

Recommendation:

As explained in the Description section.

Nibble Count in Nibble Parser May Exceed the Maximum Allowable Value

Project	Gear Bridges - ZK - Prover
Type	IMPLEMENTATION
Severity	HIGH
Impact	MEDIUM
Exploitability	HIGH
Status	RESOLVED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/8

Involved Artifacts:

- prover/src/storage_inclusion/storage_trie_proof/node_parser/nibble_parser.rs

Description:

The Header Parser extracts the nibble count from the first byte of the node data, which follows a prefix that uses 2 or 3 bits. If the nibble count exceeds the maximum value that can be represented by 5 or 6 bits in the first byte, it is encoded in the second byte as well. This method supports a maximum nibble count of `MAX_STORAGE_ADDRESS_LENGTH_IN_NIBBLES = 64`. Afterwards, the Nibble Parser reads the number of nibbles specified by the Header Parser. This ensures that the correct amount of nibbles is parsed based on the previously determined nibble count. However, there is no constraint to prevent the nibble count from exceeding the maximum allowable value.

Problem Scenario:

- Let's assume that the nibble count bits in the first byte of the node data are set to their maximum value.
- In this case, the Header Parser will read the second byte of the node data to compute the nibble count, which provides sufficient space to store a value exceeding `MAX_STORAGE_ADDRESS_LENGTH_IN_NIBBLES`.
- The Nibble Parser receives nibble count as an input parameter that exceeds the maximum allowable value likely indicates that the encoded node data is incorrect or corrupted.

Recommendation:

A constraint enforcing the maximum allowable value for the nibble count should be added to the Nibble Parser circuit.

Inadequate Error Handling May Cause Outages and Halt the Proof Generation Process

Project	Gear Bridges - ZK - Prover
Type	IMPLEMENTATION
Severity	HIGH / MEDIUM [1]
Impact	HIGH
Exploitability	HIGH / LOW [1]
Status	ACKNOWLEDGED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/15

Involved Artifacts:

- prover/src/common/mod.rs
- prover/src/block_finality/validator_signs_chain/mod.rs

Description:

The current implementation of error handling within the prover component raises several concerns that could lead to significant problems.

- Generally, it is not a good practice to allow a panic to occur without proper handling.
- This can have a direct impact on how validators operate. If a single thread responsible for generating a proof for one validator fails, it could cause the entire proof generation process to halt. This means that if one validator fails to sign a message or panics during the proof generation process, the failure could cascade, leading to the failure of all other validators in the process, even though there might be more that 2/3 of them that are valid. Such a scenario not only disrupts the operation of the network but also raises concerns about the system's ability to handle individual failures without impacting the whole.
- It opens the door for exploitation by malicious actors since validator with malicious intent could intentionally trigger errors that lead to a panic, repeatedly crashing the process and causing significant disruption. This vulnerability could be exploited to delay or completely block the proof generation process, undermining the reliability and security of the system. Implementing more resilient error handling would help mitigate this risk by isolating errors and preventing a single point of failure from affecting the entire system.

Problem Scenario:

- The prove_from_builder() and prove_from_circuit_data() functions rely on the prove() function from the Plonky2 library, which returns a `Result` type that includes an error if one occurs. Unfortunately, potential errors are not properly handled within the `prove_from_builder()` function, as the unwrap() function is used. This approach is problematic because `unwrap()` can cause a panic when an error is encountered, leading to unexpected crashes that could prevent the Prover component from functioning properly.
- Separating the job into threads when generating an Indexed Validator Sign proof is generally a good practice. However, leaving it without proper handling of panics can lead to serious problems. When

using the `rayon-core` library for multithreading, it's crucial to protect against panics within thread processing. Without proper error handling, a single thread could propagate a panic and bring the entire program to a halt.

Recommendation:

- A better practice would be to use pattern matching or an if-else statement to explicitly handle the `Err` case. By propagating the error back, the Prover can track and handle it appropriately, ensuring that the system remains stable even when errors occur.
- Wrapping the `panic::catch_unwind()` call inside the `thread_pool.scope()` function can help prevent panics from propagating, allowing for proper handling afterward. Here is the example:

```
pre_commits.into_par_iter().enumerate().for_each_with(
        sender,
        |sender, (id, pre_commit)| {
            thread_pool.scope(|_| {
                // capturing the cause of an unwinding panic if one occurs
                let result = panic::catch_unwind(|| {
                    let proof = IndexedValidatorSign {
                        public_key: pre_commit.public_key.clone(),
                        index: pre_commit.index,
                        signature: pre_commit.signature.clone(),
                        message: pre_commit.message.clone(),
                    }
                    .prove();

                    sender
                        .send((id, proof))
                        .expect("Failed to send proof over channel");
                });
                // handling of the caught panic
                if let Err(err) = result {
                    println!("Thread panicked while processing pre_commit {}: {:?}",
id, err);
                }
            });
        },
    );
```

Additional Observation:

While this issue is marked as high severity within the scope of the audit due to the risk of compromising the prover component's integrity if it fails to defend against errors or attacks, and given the prover's importance in the system, it is important to note that, within the context of the use case, it may not pose a significant risk because all components are intended to be controlled by Gear. During discussions on this issue, Gear assured us that the relayer is expected to always submit valid signatures. While we value their assurance, our analysis and the high severity rating are based on a broader assessment of potential risks, independent of this guarantee.

Point Decompression Can Cause the Code to Panic

Project	Gear Bridges - ZK - Circuits
Type	IMPLEMENTATION
Severity	HIGH / MEDIUM [2]
Impact	HIGH
Exploitability	HIGH / LOW [2]
Status	ACKNOWLEDGED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/18

Involved Artifacts:

- plonky2_ed25519/src/gadgets/eddsa.rs

Description:

Decompression of A and R should result in valid curve points. For point decompression, Gear relies on curve25519_dalek decompress function, which has appropriate input validation and will return `None` if the passed bit array doesn't represent a compressed curve point. In that case, unwrap will panic. We recommend following Rust docs and explicitly handling the `None` case.

Problem Scenario:

As explained in the Description section.

Recommendation:

As explained in the Description section.

Additional Observation:

While this issue is marked as high severity within the scope of the audit due to the risk of compromising the prover component's integrity if it fails to defend against errors or attacks, and given the prover's importance in the system, it is important to note that, within the context of the use case, it may not pose a significant risk because all components are intended to be controlled by Gear. During discussions on this issue, Gear assured us that the input for the decompress function is expected to always be valid. While we value their assurance, our analysis and the high severity rating are based on a broader assessment of potential risks, independent of this guarantee.

A Block with Enough Validator Votes on GRANDPA Message Can Be Rejected as Finalized

Project	Gear Bridges - ZK - Prover
Type	IMPLEMENTATION
Severity	HIGH / MEDIUM [3]
Impact	HIGH
Exploitability	MEDIUM / LOW [3]
Status	ACKNOWLEDGED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/6

Involved Artifacts:

- prover/src/block_finality/mod.rs
- prover/src/block_finality/validator_signs_chain/mod.rs

Description:

When creating `BlockFinality` proof `pre_commits` set of validator signers of GRANDPA message are filtered so thus a max of `processed_validator_count` (which is a minimum required number of validator votes needed to successfully prove block finalization) is sampled. These sampled pre commits are stored in `processed_pre_commits`.

This sampling method optimizes the signature check but does not guarantee that a block will be finalized correctly. Even if there are enough valid signatures, the presence of at least one Byzantine faulty signature in the pre-commits could cause the finalized block to fail verification.

Problem Scenario:

- Lets assume that `pre_commits` set has more than `processed_validator_count` number of precommits which is a minimum required number of validator votes needed to successfully prove block finalization;
- Lets assume that `pre_commits` set contains exactly one byzantine precommit (a precommit that is not valid, i.e. will not pass the signature check) and that the rest of them are valid;
- `pre_commits` is sampled for a `processed_validator_count` number of precommits ;
- Assume that among the sampled precommits is the mentioned byzantine precommit;
- This means that the sampled precommits will not have enough valid precommits in order to pass the more than 2/3 vote check even though there were more valid precommits that were not sampled from `pre_commits` set and these would be enough to pass the 2/3 vote check.

Recommendation:

A check for all precommit votes should be done to avoid the aforementioned false-positive scenario. If optimization is needed it could be implemented when enough signature checks have passed.

Additional Observation:

While this issue is marked as high severity within the scope of the audit due to the risk of compromising the prover component's integrity if it fails to defend against errors or attacks, and given the prover's importance in the system, it is important to note that, within the context of the use case, it may not pose a significant risk because all components are intended to be controlled by Gear. During discussions on this issue, Gear assured us that the relayer is expected to always submit valid signatures. While we value their assurance, our analysis and the high severity rating are based on a broader assessment of potential risks, independent of this guarantee.

Nibble Padding Should Be Set to Zero

Project	Gear Bridges - ZK - Prover
Type	IMPLEMENTATION
Severity	MEDIUM
Impact	MEDIUM
Exploitability	MEDIUM
Status	RESOLVED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/9

Involved Artifacts:

- prover/src/storage_inclusion/storage_trie_proof/node_parser/nibble_parser.rs

Description:

According to the decode_plan function, if the nibble count is odd, the first nibble is excluded from the node's partial address and serves as padding, which should be set to zero. The Nibble Parser doesn't contain a constraint to ensure that nibble padding is set to zero.

Problem Scenario:

- Let's assume that the nibble count is an odd, non-zero number. This implies that the first nibble acts as padding.
- Let's assume that the first nibble has a non-zero value.
- This scenario is possible because there is no constraint to prevent it, and it would indicate that the node data has an incorrect format or is corrupted.

Recommendation:

The Nibble Parser must verify that the first nibble is zero when the nibble count is an odd number.

Missing Range-Checks

Project	Gear Bridges - ZK - Circuits
Type	IMPLEMENTATION
Severity	LOW
Impact	MEDIUM
Exploitability	LOW
Status	RESOLVED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/4

Involved Artifacts:

- plonky2_ecdsa/src/gadgets/biguint.rs
- plonky2_ed25519/src/gadgets/nonnative.rs

Description:

For consistency with other functions, we recommend adding range checks in the following non-native arithmetic functions:

1. add_nonnative - sum and overflow should be checked as difference and overflow in sub_nonnative function;
2. inv_nonnative - div and inv should be checked as e.g. difference or product.

Additionally, in function div_rem_biguint:

1. div and rem could also be range-checked here.
2. More importantly, rem should be strictly less than b; so instead of the rem<=b check it would be better to check if b<=rem is false. This way cases like (a, b, q, r) = (6, 2, 2, 2) will be avoided.
3. And finally, this statement will cause overflow if `b_len == a_len+1`; so either the operation order should be changed to `a_len+1 - b_len` or the if statement to `b_len > a_len`.

Problem Scenario:

As explained in the Description section.

Recommendation:

As explained in the Description section.

Issues Related to Base and Scalar Field Implementations

Project	Gear Bridges - ZK - Circuits
Type	IMPLEMENTATION
Severity	LOW
Impact	LOW
Exploitability	MEDIUM
Status	RESOLVED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/3

Involved Artifacts:

plonky2_ed25519/src/field/ed25519_base.rs
plonky2_ed25519/src/field/ed25519_scalar.rs

Description:

Base field of an elliptic curve is the field over which the curve is defined. Curve parameters and all coordinates must be elements of the base field. The base field order of edwards25519 curve is $p=2^{255}-19$.

Scalar field is the field of scalars used in the operations performed on the curve (e.g. scalar multiplication, and pairing). The scalar field size is also the size of the largest subgroup of prime order, which is $r = 2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed$ for edwards25519 curve.

Both fields are implementing Field trait from plonky2 library, hence have to, among others, set values for constants mentioned below:

1. Constant TWO_ADICITY[1] should be 2 (not 1) both in base and scalar fields because the order of both multiplicative groups is divisible by 4 (so there exists a multiplicative subgroup of order $4=2^2$), and is not divisible by any larger power of two.
2. Constant CHARACTERISTIC_TWO_ADICITY (defined both in base and scalar fields) can either remain equal to TWO_ADICITY (it is mathematically correct, because the field is of prime order, and hence its characteristic is equal to its order), or it can be set to None according to this comment.
3. Constant NEG_ONE in scalar field is set to field order+1 instead of field order-1.

Additionally, all the field elements must be less then the field order, which is why conversion of BigUint to field element is missing a range check (or reduction modulo field order) for both in base and scalar field.

([1]Definition of TWO_ADICITY can be found in, e.g., this paper)

Problem Scenario:

As explained in the Description section.

Recommendation:

As explained in the Description section.

Issues Related to Twisted Edwards Curve Implementation

Project	Gear Bridges - ZK - Circuits
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	MEDIUM
Status	RESOLVED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/2

Involved Artifacts:

plonky2_ed25519/src/curve/curve_adds.rs
plonky2_ed25519/src/curve/curve_types.rs
plonky2_ed25519/src/curve/ed25519.rs
plonky2_ed25519/src/gadgets/curve.rs

Description:

Twisted Edwards curves are represented with the following equation:

$$Ax^2 + y^2 = 1 + Dx^2y^2,$$

where A and D are distinct non-zero elements (i.e., A*D*(A-D) != 0).

Curve equation presented in the projective form is:

$$(A*X^2 + Y^2)Z^2 = Z^4 + D*X^2*Y^2.$$

Neutral element for this curve in affine coordinates is (0, 1), while in projective coordinates (0, 1, 1). The inverse for a point (x, y) is calculated as (-x, y).

Edwards25519 is a Twisted Edwards curve with A = -1 and D = -121665/121666 defined over the field of prime order $p = 2^{255} - 19$.

Twisted Edwards and Edwards25519 are described in the following references:

- Twisted Edwards Curves
- EFD/Twisted Edwards
- High-speed high-security signatures (ed25519 paper)
- RFC 8032: EdDSA
- RFC 7748: Elliptic Curves for Security

Gear's implementation of base type that represents Twisted Edwards curves and its instance for edwards25519 curve does not match the aforementioned referenced literature in the following places:

1. Incorrect curve equation.
    a. function is_safe_curve - this function checks the discriminant of Weierstrass curve instead of A*D*(A-D) != 0;
    b. function is_valid checks $y^2 == x^2 + D *x^2*y^2 + A$ instead of the curve equation: $Ax^2 + y^2 = 1 + Dx^2y^2$;

   c. function curve_assert_valid - the following check is not correct:
      i. $y^2 = a + x^2 + b*x^2*y^2$; it is not aligned with curve equation ($Ax^2 + y^2 = 1 + Dx^2y^2$).
2. z-coordinate != 0 checks.
   z coordinate will never be equal to zero for Twisted Edwards curves, so the checks are unnecessary. They appear in:
   a. plonky2_ed25519/src/curve/curve_types.rs: here, here, here and here;
   b. plonky2_ed25519/src/curve/curve_adds.rs: here and here.
3. The neutral element and the zero-flag.
   As mentioned, the neutral element for this curve in affine coordinates is (0, 1), while in projective coordinates (0, 1, 1) . The zero-flag is excessive and should be removed:
   a. const ZERO - this constant represents the neutral element, so the coordinates should be x=0, y=1 (and the zero flag is not necessary);
   b. function to_projective - when converting the neutral element in affine coordinates to projective coordinates, the z coordinate should equal 1, not 0;
   c. It is not correct that doubling will always return non-zero points, for example: (0, -1) + (0, -1) = (0, 1);
   d. PartialEq - the neutral element is a valid curve point, so the equality check can be done as with other points, i.e., this check is sufficient;
   e. const ZERO - this constant represents the neutral element, so the coordinates should be x=0, y=1, z=1;
   f. In plonky2_ed25519/src/curve/curve_adds.rs: if this check were removed, it would be possible to get the neutral element as a result of point addition..
4. Inverses.
   The inverse for a point (x, y) is (-x, y). Incorrect definition of an inverse, namely (x, -y), appears in:
   a. plonky2_ed25519/src/curve/curve_adds.rs: here and here
5. Addition formulas in plonky2_ed25519/src/curve/curve_adds.rs:
   a. This addition formula holds for Weierstrass curves, not twisted Edwards curves (https://www.hyperelliptic.org/EFD/g1p/data/shortw/projective/addition/madd-1998-cmo);
   b. This check is not necessary since addition and doubling formulas are the same for affine points.
6. Constant A.
   a. plonky2_ed25519/src/curve/ed25519.rs - const A is set to 1, and it should be set to -1 if the curve equation is written in the standard form;
   b. We recommend using constant A instead of hard coding its value (-1) in all the formulas, e.g., in plonky2_ed25519/src/curve/curve_types.rs (here, here and here) or in plonky2_ed25519/src/curve/curve_adds.rs (here and here) or in plonky2_ed25519/src/gadgets/curve.rs (here and here).

Problem Scenario:

As explained in the Description section.

Recommendation:

As explained in the Description section.

Relayer Smart Contract Unnecessarily Verifies Already Verified and Stored Merkle Roots with Their ZK Proofs

Project	Gear Bridges - ZK - Ethereum Smart Contracts
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	LOW
Status	ACKNOWLEDGED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/21

Involved Artifacts:

- ethereum/src/Relayer.sol

Description:

Even if a zk proof for a specific block number and Merkle root has already been successfully verified and those two stored in the `_block_numbers` and `_merkle_roots` maps, submitting these again will lead to the same process of verification and storage unnecessarily.

Problem Scenario:

Unnecessary call to gnark's verification of the provided zk proof is done in these situations.

Recommendation:

It is recommended that a check if the incoming `block_number` and `merkle_root` are already stored in the `block_numbers` and `_merkle_roots` maps is introduced at the beginning of the `submitMerkleRoot` function.

Usage of Unaudited and Partially Audited Libraries

Project	Gear Bridges - ZK
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue(s)	

Involved Artifacts:

- circuits
- prover
- ethereum

Description:

The entire project relies on libraries that have not yet been fully audited, as indicated in the disclaimers available on their GitHub repositories. Specifically, Plonky2 has not been audited at all, while gnark has only undergone a partial audit. The use of these libraries in a production environment may potentially introduce unknown vulnerabilities.

Code Improvements

Project	Gear Bridges - ZK - Prover
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue(s)	https://github.com/mertwole/gear-bridges-audit/issues/17 https://github.com/mertwole/gear-bridges-audit/issues/7 https://github.com/mertwole/gear-bridges-audit/issues/16

Involved Artifacts:

- prover/src/block_finality/mod.rs
- prover/src/storage_inclusion/storage_trie_proof/node_parser/leaf_parser/inlined_data_parser.rs
- prover/src/common/mod.rs
- prover/src/common/targets/array.rs

Description:

- The process of calculating the sufficient validator count (> 2/3) can be simplified by adapting the formula to:
  `(2 * self.validator_set.len()) / 3 + 1`, instead of
  `2 * (self.validator_set.len() / 3) + 1` and
  `2 * (self.validator_set.len() - 2) / 3 + 2`.
  This approach performs the multiplication first, then the division, and finally adds one, which not only eliminates the need for a separate implementation of the second option, but also achieves greater consistency within the code.
- The struct `InlindedDataParserInputTarget` likely contains a typo. It should be named `InlinedDataParserInputTarget` to match the common spelling of "inlined."
- In the following function, consider the optimization, which directly creates an array and assigns each bit value in a loop, which can be more efficient and avoids the need for intermediate vector allocation and conversion.
  Another way to write this function in Rust, avoiding the use of Vec and try_into, could be:

```
pub fn byte_to_bits(byte: u8) -> [bool; 8] {
    let mut bits = [false; 8];
    for i in 0..8 {
        bits[7 - i] = (byte >> i) & 1 == 1;
    }
    bits
}
```

- Within the `constant_read_array` function, the current implementation is fairly straightforward and concise, but there are a few possible improvements.
  Since you know the size of the array at compile-time, you can directly build the array without the need for allocating a `Vec` and then converting it into an array with `try_into`, which can help in optimizing performance.
  By using static arrays instead of dynamic ones, stack memory is used instead of heap memory, which gives an advantage in performance.

```
pub fn constant_read_array<const R: usize>(&self, at: usize) -> ArrayTarget<T,
R> {
    let mut array = [self.constant_read(at); R];
    for i in 1..R {
        array[i] = self.constant_read(at + i);
    }
    ArrayTarget(array)
}
```

Appendix: 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.

Finding Severity Categories

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

		EXPLOITABILITY			
		NONE	**LOW**	**MEDIUM**	**HIGH**
IMPACT	**NONE**	INFORMATIONAL	INFORMATIONAL	INFORMATIONAL	INFORMATIONAL
	LOW	INFORMATIONAL	LOW	LOW	MEDIUM
	MEDIUM	INFORMATIONAL	LOW	MEDIUM	HIGH
	HIGH	INFORMATIONAL	MEDIUM	HIGH	HIGH

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.