Linea Contracts Update December 2023

Date	January 2024
Auditors	Rai Yang

1 Executive Summary

This report presents the results of our engagement with **Linea** to review **an update of their smart contracts**, in particular the rollup and message service contracts.

The review was conducted primarily by **Rai Yang**, with some support from **Heiko Fisch** and **George Kobakhidze**. It took place between **December 11, 2023**, and **January 12, 2024**.

Linea contracts have undergone several audits by Consensys Diligence (Verifier, Message Service, Canonical Token Bridge, Cross-Chain Governance Executor, Custom Bridged Token) and OpenZeppelin (Verifier, Bridge) throughout 2023.

Recently, in preparation for EIP-4844, the Linea team has implemented an interim upgrade for compressed data submission and finalization and made some changes to the message service; these changes are the subject of the current review.

2 Scope

The scope for this audit has been defined in a document provided by the Linea team; it describes the changes in detail, explains the motivation, and lists the contracts that were altered. This document served as a reference for this engagement. A short outline of the changes is given in section 3.

We started our review at the commit hash `0154676f95a510a855622f8ac9b07816f94edf08` of the GitHub repository `Consensys/linea-contracts-audit`. Early in the engagement, two changes were incorporated into the codebase, which led to the commit hash `a4fb9a48dffc2bc2fb5f9161b2dde19c2bafc5e6`. After another small change in the last week, the final version of the codebase considered in this audit has the commit hash `bb6eb7284d1ac9574dc69e654abe5ccb8d8ded1a`. A list of all Solidity files (except tests and mocks) in the repository at the commit hash `bb6eb7284d1ac9574dc69e654abe5ccb8d8ded1a`, together with their SHA-1 hashes, can be found in Appendix 1.

We have reviewed the issues and fixes for the OpenZeppelin report in commit 075b26e9656afa11197ebc2377f593d2cc1db26b, we agree M01-M11, L1, L2, L4-L8, L11, N1-N7, N9-N12 are valid issues and M01-M10, L1, L2, L4-L8, N1-N7, N9-N12 are fixed or addressed. However for M01,M06 and M08, we believe current fix by adding checks for block number order in the contract is not that helpful to resolve the issue, since the operator can always submit an incorrect block number and the check complicates the logic, adds gas cost and may introduce errors, ultimately the block number order are proved by the prover and verified in the data finalization, therefore we recommend to remove these checks.

3 Summary of the Changes

The motivation for these contract changes is the highly anticipated EIP-4844 (also known as "proto-danksharding"), which will bring enormous benefits, especially for rollups. Once implemented, this improvement proposal will provide much cheaper data storage in so-called blobs. While blob data is not stored permanently, it is kept for long enough to give everyone a chance to grab it – what is commonly referred to as "data availability." As rollups need to publish relatively large amounts of data on L1 – which is currently achieved via calldata – blobs will drastically reduce the L1 fees rollups have to pay.

In order to keep the necessary adaptations to utilize blobs to a minimum when proto-danksharding goes live, the Linea team has implemented an interim update that behaves similarly to EIP-4844 but employs compressed calldata instead of blobs. More precisely – and quoting from Linea's description – the changes that have been implemented and are the subject of this review are:

1. Compressed data submission. This includes validation of the data, securing access to functionality, as well as public input generation for the proof finalization.
2. Access-controlled block finalization, hashing the provided compressed data, as well as other calldata for public input generation for the proof, and then proof verification.
3. Block finalization without proof verification.
4. L1 to L2 message flow, computing a rolling hash (each message added is hashed with the previous) for L1 for message addition, and a mirrored calculation on L2 when anchoring, where the feedback loop sends the latest rolling hash anchored to L1 in finalization to validate no censorship or tampering.
5. L2 to L1 message flow inclusive of L1 anchoring in the finalization using indexed sparse Merkle trees on L2 for message addition and Merkle proofs for message validation on L1.

4 Security Specification

This section describes, from a security perspective, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

4.1 Actors

The relevant actors are listed below with their respective abilities:

- User: Use the message service on L1 and L2.
- Coordinator: Moves information between L1 and L2. This includes submitting and finalizing compressed block data and relaying messages between L1 and L2.
- Sequencer: Builds and executes L2 blocks, generates execution trace and conflated block data.
- Prover: Proves correct state transitions and block data execution with the execution trace provided by the sequencer.
- Security Council: A Linea/Consensys-controlled multi-sig wallet that deploys contracts and performs upgrades.
- Postman: Linea's off-chain message delivery service or third-party service delivering the messages to claim the delivery fee. Each postman is not dependent on the other to function.

4.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- The single coordinator relays messages between L1 and L2 correctly and timely and does not censor L1 → L2 or L2 → L1 messages.
- The single sequencer sequences the L2 transaction correctly, timely and does not censor L2 messages.
- The single prover proves the execution correctly, sets `chainID` correctly in the circuit for the corresponding chain, generates Merkle roots correctly, and does not censor L2 → L1 messages.
- Postman delivers the messages to the destination layer correctly and rationally based on the fee and gas cost of delivering the message.
- Security Council performs the service administration properly and upgrades the contracts correctly and securely.

4.3 Security Properties

The following is a non-exhaustive list of security properties that were considered in this audit:

- Storage layout is not broken.
- Reentrancy issues have not been introduced.
- Previous issues have not been reintroduced.
- Access control is still in place and has been used correctly.
- Attack vectors have not been introduced.
- Backwards compatibility remains.
- Duplicate claiming is not possible.
- If external contracts are used, security and lack of exploitability is maintained.
- Compressed data item submission is validated correctly.
- Messaging system is sound.
- Proof verification is sound (includes rollup mechanism and public input generation).
- The algorithm (proof of equivalence) to check compressed data submitted to L1 is the same data used in the prover is correctly implemented.
- New (if used) messaging mechanism is sound and has no attack/manipulation vectors.
- Merkle root proving is valid.
- If submitted data is incorrect in finalization, operator can resubmit and finalize data.
- L1 contract migration/update is sound once EIP-4844 is implemented on L1.
- L1 → L2 message claiming refund subsdize(`REFUND_OVERHEAD_IN_GAS`) is correctly set that prevents user gas price manipulation.
- L1 → L2 messages are not tampered by the coordinator.
- Linea chain can fork when L1 forks.
- L1 contract can distinguish data submitted from different Linea forks.

5 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 Incorrect Final Block Number Can Be Finalized Critical ✓ Fixed

Resolution

fixed by adding a recommended check of `finalBlockNumber` matching the last block number of the submitted data in `_finalizeCompressedBlocks` and a check in the prover and adding `finalBlockNumber` and `lastFinalizedBlockNumber` in the public input of the verifier in the finalization in PR-24

Description

In the data finalization function `finalizeCompressedBlocksWithProof`, `finalizationData.finalBlockNumber` is the final block number of the compressed block data to be finalized. However, there is no check in the contract or the prover to ensure `finalBlockNumber` is correct when there is no new data submitted in the finalization, i.e., `submissionDataLength == 0`. The prover can submit an incorrect final block number and, as a result, the finalized block number (`currentL2BlockNumber`) would be incorrect. Consequently, the prover can skip block data in the finalization.

Examples

contracts/LineaRollup.sol:L347

```
currentL2BlockNumber = _finalizationData.finalBlockNumber;
```

contracts/LineaRollup.sol:L199-L201

```
if (stateRootHashes[currentL2BlockNumber] != _finalizationData.parentStateRootHash) {
  revert StartingRootHashDoesNotMatch();
}
```

Recommendation

In `_finalizeCompressedBlocks` , check if `finalBlockNumber` is equal to the last block number (`finalBlockInData`) of the last item of submitted block data. Another solution is to have the prover show that `finalBlockNumber` is correct in the proof by providing the last finalized block number (`lastFinalizedBlockNumber`) and verify it by adding `finalBlockNumber` and `lastFinalizedBlockNumber` in the public input of the verifier in the finalization.

5.2 Finalization Fails for the First Batch of Data Submitted After Migration to the Updated Contract

Critical ✓ Fixed

Resolution

Linea responded:

> Our migration and deployment strategy is to have as close to zero downtime as possible. Because of this, implementing the recommendation of Set the correct initial value for dataFinalStateRootHashes for the initial batch of compressed block data. is going to be difficult as we won't know it at the time (L2 block and state is indeterminable in advance) and would require a pausing of the contracts.

The issue is fixed in PR-24 by wrapping the check

```
 if (startingParentFinalStateRootHash != _finalizationData.parentStateRootHash) {
          revert FinalStateRootHashDoesNotMatch(
            startingParentFinalStateRootHash,
            _finalizationData.parentStateRootHash
          );
  }
```

with `if (startingDataParentHash != EMPTY_HASH) {` to allow it go through the first time after migration, subsequent finalization will be checked.

Description

When submitting the initial batch of compressed block data after the contract update, the finalization will fail.

In function `_finalizeCompressedBlocks` , `startingDataParentHash = dataParents[_finalizationData.dataHashes[0]]` will be empty and, therefore, `startingParentFinalStateRootHash = dataFinalStateRootHashes[startingDataParentHash]` will be empty too. The check `_finalizationData.parentStateRootHash == stateRootHashes[currentL2BlockNumber]` requires `_finalizationData.parentStateRootHash == _initialStateRootHash` , which is not empty, so the condition `startingParentFinalStateRootHash != _finalizationData.parentStateRootHash` is true, and we revert with the error `FinalStateRootHashDoesNotMatch` :

contracts/LineaRollup.sol:L199-L201

```
if (stateRootHashes[currentL2BlockNumber] != _finalizationData.parentStateRootHash) {
  revert StartingRootHashDoesNotMatch();
}
```

contracts/LineaRollup.sol:L283-L294

```
if (finalizationDataDataHashesLength != 0) {
  bytes32 startingDataParentHash = dataParents[_finalizationData.dataHashes[0]];

  if (startingDataParentHash != _finalizationData.dataParentHash) {
    revert ParentHashesDoesNotMatch(startingDataParentHash, _finalizationData.dataParentHash);
  }

  bytes32 startingParentFinalStateRootHash = dataFinalStateRootHashes[startingDataParentHash];

  if (startingParentFinalStateRootHash != _finalizationData.parentStateRootHash) {
    revert FinalStateRootHashDoesNotMatch(startingParentFinalStateRootHash, _finalizationData.parentStateRootHash);
  }
```

Recommendation

Set the correct initial value for `dataFinalStateRootHashes` for the initial batch of compressed block data.

5.3 Prover Can Censor L2 → L1 Messages Major Partially Addressed

Resolution

Linea responded that the prover enforces all messages are included in the circuit, however with the circuit code is not opensourced yet, this still need to be verified

Description

In L2 → L1 messaging, messages are grouped and added to a Merkle tree by the prover. During finalization, the operator (coordinator) submits the Merkle root to L1, and the user SDK rebuilds the tree to which the message is added and generates a Merkle proof to claim against the root finalized on L1. However, the prover can skip messages when building the tree. Consequently, the user cannot claim the skipped message, which might result in frozen funds.

Currently, the prover is a single entity owned by Linea. Hence, this would require malice or negligence on Linea's part.

Examples

contracts/LineaRollup.sol:L314-L315

```
_addL2MerkleRoots(_finalizationData.l2MerkleRoots, _finalizationData.l2MerkleTreesDepth);
_anchorL2MessagingBlocks(_finalizationData.l2MessagingBlocksOffsets, lastFinalizedBlock);
```

Recommendation

Decentralize the prover, so messages can be included by different provers.

5.4 Malicious Operator Might Finalize Data From a Forked Linea Chain Major ✓ Fixed

Resolution

Linea team responded that `chainId` is hard coded in the prover's circuit, the verifier key (verifier contract) would be different for different chainId, the proof won't pass the verification unless the forked Linea chain has the same `chainId` as the canonical chain

Description

A malicious operator (prover) can add and finalize block data from a forked Linea chain, so transactions on the forked chain can be finalized, causing a loss of funds from the L1.

For example, a malicious operator forks the canonical chain, then the attacker sends the forked chain Ether to L1 with `sendMessage` from the forked L2. The operator then submits the block data to L1 and finalizes it with `finalizeCompressedBlocksWithProof`, using the finalization data and proof from the forked chain. (Note that the malicious prover sets the forked chain `chainId` in its circuit as a constant.) The L1 contract (`LineaRollup`) doesn't know whether the data and the proof are from the canonical L2 or the forked one. The finalization succeeds, and the attacker can claim the bridged forked chain Ether and steal funds from L1.

As there is currently only one operator and it is owned by the Linea team, this kind of attack is unlikely to happen. However, when the operator and the coordinator are decentralized, the likelihood of this attack increases.

Examples

contracts/LineaRollup.sol:L211-L222

```
uint256 publicInput = uint256(
  keccak256(
    abi.encode(
      shnarf,
      _finalizationData.parentStateRootHash,
      _finalizationData.lastFinalizedTimestamp,
      _finalizationData.finalBlockNumber,
      _finalizationData.finalTimestamp,
      _finalizationData.l1RollingHash,
      _finalizationData.l1RollingHashMessageNumber,
      keccak256(abi.encodePacked(_finalizationData.l2MerkleRoots))
    )
```

contracts/LineaRollup.sol:L314

```
_addL2MerkleRoots(_finalizationData.l2MerkleRoots, _finalizationData.l2MerkleTreesDepth);
```

Recommendation

Add `chainId` in the `FinalizationData` as a public input of the verifier function `_verifyProof`, so the proof from the forked Linea chain will not pass the verification because the `chainId` won't match.

5.5 The Compressed Block Data Is Not Verified Against Data in the Prover During Data Submission

Medium Acknowledged

Resolution

Linea has acknowledged this issue and will implement the recommended check with the EIP-4844 upgrade using the KZG precompile

Description

When the sequencer submits the batched block data with the `submitData` function, it's expected to check that the submitted commitment of the compressed block data `keccak(_submissionData.compressedData)` and the commitment of the block data used in the prover (`snarkHash`) commit to the same data. This is done by proof of equivalence; the `x` is calculated by hashing `keccak(_submissionData.compressedData)` and `snarkHash` , and `y` is provided by the prover. Then it's verified that `P(x) = y` , where `P` is a polynomial that encodes the compressed data (`_submissionData.compressedData`). However, in the `submitData` function, `y` is evaluated by `_calculateY` but it is not checked against the `y` provided by the prover. In fact, the prover doesn't provide `y` to the function; instead `x` and `y` are provided to the prover who would evaluate `y'` and compare it with `y` from the contract, then `x` and `y` are included in the public input for the proof verification in the finalization.

```
shnarf = keccak256(
      abi.encode(
        shnarf,
        _submissionData.snarkHash,
        _submissionData.finalStateRootHash,
        compressedDataComputedX,
        _calculateY(_submissionData.compressedData, compressedDataComputedX)
      )
 );
```

The only difference is if the two commitments don't commit to the same block data (meaning the data submitted doesn't match the data used in the prover), `submitData` would fail – while in the current implementation, it would fail in the proof verification during the finalization. As a result, if the data submitted doesn't match the data in the prover in the finalization, the operator has to submit the correct data again in order to finalize it. Linea stated they will verify it in the data submission, once EIP-4844 is implemented.

Examples

contracts/LineaRollup.sol:L131-L173

```
function _submitData(SubmissionData calldata _submissionData) internal returns (bytes32 shnarf) {
  shnarf = dataShnarfHashes[_submissionData.dataParentHash];

  bytes32 parentFinalStateRootHash = dataFinalStateRootHashes[_submissionData.dataParentHash];
  uint256 lastFinalizedBlock = currentL2BlockNumber;

  if (_submissionData.firstBlockInData <= lastFinalizedBlock) {
    revert FirstBlockLessThanOrEqualToLastFinalizedBlock(_submissionData.firstBlockInData, lastFinalizedBlock);
  }

  if (_submissionData.firstBlockInData > _submissionData.finalBlockInData) {
    revert FirstBlockGreaterThanFinalBlock(_submissionData.firstBlockInData, _submissionData.finalBlockInData);
  }

  if (_submissionData.parentStateRootHash != parentFinalStateRootHash) {
    revert StateRootHashInvalid(parentFinalStateRootHash, _submissionData.parentStateRootHash);
  }

  bytes32 currentDataHash = keccak256(_submissionData.compressedData);

  if (dataFinalStateRootHashes[currentDataHash] != EMPTY_HASH) {
    revert DataAlreadySubmitted(currentDataHash);
  }

  dataParents[currentDataHash] = _submissionData.dataParentHash;
  dataFinalStateRootHashes[currentDataHash] = _submissionData.finalStateRootHash;

  bytes32 compressedDataComputedX = keccak256(abi.encode(_submissionData.snarkHash, currentDataHash));

  shnarf = keccak256(
    abi.encode(
      shnarf,
      _submissionData.snarkHash,
      _submissionData.finalStateRootHash,
      compressedDataComputedX,
      _calculateY(_submissionData.compressedData, compressedDataComputedX)
    )
  );

  dataShnarfHashes[currentDataHash] = shnarf;

  emit DataSubmitted(currentDataHash, _submissionData.firstBlockInData, _submissionData.finalBlockInData);
}
```

contracts/LineaRollup.sol:L384-L413

```
function _calculateY(
  bytes calldata _data,
  bytes32 _compressedDataComputedX
) internal pure returns (bytes32 compressedDataComputedY) {
  if (_data.length % 0x20 != 0) {
    revert BytesLengthNotMultipleOf32();
  }

  bytes4 errorSelector = ILineaRollup.FirstByteIsNotZero.selector;
  assembly {
    for {
      let i := _data.length
    } gt(i, 0) {

    } {
      i := sub(i, 0x20)
      let chunk := calldataload(add(_data.offset, i))
      if iszero(iszero(and(chunk, 0xFF00000000000000000000000000000000000000000000000000000000000000))) {
        let ptr := mload(0x40)
        mstore(ptr, errorSelector)
        revert(ptr, 0x4)
      }
      compressedDataComputedY := addmod(
        mulmod(compressedDataComputedY, _compressedDataComputedX, Y_MODULUS),
        chunk,
        Y_MODULUS
      )
    }
  }
}
```

Recommendation

Add the compressed block data verification in the `submitData` function.

5.6 Empty Compressed Data Allowed in Data Submission Medium ✓ Fixed

Resolution

fixed by adding a recommended check in PR-20

Description

In `submitData`, the coordinator can submit data with empty `compressedData` in `_submissionData`, which is not a desired purpose of this function and may cause undefined system behavior.

Examples

contracts/LineaRollup.sol:L115-L124

```
function submitData(
  SubmissionData calldata _submissionData
)
  external
  whenTypeNotPaused(PROVING_SYSTEM_PAUSE_TYPE)
  whenTypeNotPaused(GENERAL_PAUSE_TYPE)
  onlyRole(OPERATOR_ROLE)
{
  _submitData(_submissionData);
}
```

Recommendation

Add a check to disallow data submission with empty `compressedData`.

Appendix 1 - Solidity files with their SHA-1 hashes

The following table contains all Solidity files (with the exception of tests and mocks) in the client's repository with their corresponding SHA-1 hash; the repository is viewed at `bb6eb7284d1ac9574dc69e654abe5ccb8d8ded1a` – which is the final version of the codebase considered for this audit. Not all these files have changed since earlier audits, and we have only reviewed the changes that the client outlined in the audit companion document.

File	SHA-1 hash
contracts/LineaRollup.sol	b582343b36a478bd932174818de0803e6c5572bb
contracts/LineaRollupInit.sol	9b5f88010122d812b50edca5338fcd5962d50b6d
contracts/ProxyAdminReplica.sol	29ae784a508bc109a73bdec99b7c4ca0f329952b
contracts/ZkEvmV2.sol	368f8c1722e5186afabc15bff2a337a3a16bba33
contracts/interfaces/IGenericErrors.sol	7c4ad94a42172cce1dfae7c48107e236d19f12ae
contracts/interfaces/IMessageService.sol	5b6ef457c8c39ea63260f7784d1b12dcc78f917f
contracts/interfaces/IPauseManager.sol	eabd8700b23f5c990b4242fb45c4931038da9f1a
contracts/interfaces/IRateLimiter.sol	2b48689f1a1486e8c147a2419aef8f51aa4d3339

File	SHA-1 hash
contracts/interfaces/l1/IL1MessageManager.sol	fc82d27c0d88d3554c3a992686ad3b9ecb200087
contracts/interfaces/l1/IL1MessageManagerV1.sol	70746ddee4c0a124f7037ff6ae636062332b4417
contracts/interfaces/l1/IL1MessageService.sol	55bda614cafb2a69698a7b7593d179d86dc498a1
contracts/interfaces/l1/ILineaRollup.sol	0d87a6939123c07624911eb9988e957b26e848ad
contracts/interfaces/l1/IPlonkVerifier.sol	6b16ede6dc7c9425759b3dd1c3ca20efbe7f05c8
contracts/interfaces/l1/IZkEvmV2.sol	f0f7fff754e0129702b44fe5398ed95c1ce4506b
contracts/interfaces/l2/IL2MessageManager.sol	1a191c8bf9f90adc1738e1dc5a981b9bc559d825
contracts/interfaces/l2/IL2MessageManagerV1.sol	868c949791c3591d4fb65c4012704a4ba17ff2e4
contracts/lib/Mimc.sol	8483bfa96423fc056b979e734da9665fc81baa8d
contracts/lib/SparseMerkleProof.sol	8701164e733ae55d74e3ac35509add395d1ec5e5
contracts/lib/Utils.sol	92049b0aa15656f8ae2618b81a8e7473017c01d3
contracts/messageService/MessageServiceBase.sol	435a53352a057743f309dec8613adb0356a103b2
contracts/messageService/l1/L1MessageManager.sol	6fcb299bed71359604eff232a4cc126df8ea17ae
contracts/messageService/l1/L1MessageService.sol	2293fcf747f06645da77dfa7d25ff189f99e9814
contracts/messageService/l1/v1/L1MessageManagerV1.sol	5fa27e7baf1d0593ac7eeaf97c8784f5221eabe2
contracts/messageService/l1/v1/L1MessageServiceV1.sol	c80a35f3f47f54f9a8ad395dae947b9538315e60
contracts/messageService/l2/L2MessageManager.sol	6d264733142f53d92901b7cdb761ebdcc3b0a67e
contracts/messageService/l2/L2MessageService.sol	226070aa164edbb4934b3e4bb1c7a5a46372b0aa
contracts/messageService/l2/v1/L2MessageManagerV1.sol	0e7ad2a5a91b63bec0f4476fd3d7979973a7a0dc
contracts/messageService/l2/v1/L2MessageServiceV1.sol	1a240c77c5c1855a35a05050fa773d2752dd40e4
contracts/messageService/lib/Codec.sol	91d8caac050abde46b666961e525ea6ae96c60f7
contracts/messageService/lib/PauseManager.sol	16842973294e2083ce78cece34c01438ed6b007d
contracts/messageService/lib/RateLimiter.sol	015d90ed0c3706a265aa0d58cc55f10a4eb51009
contracts/messageService/lib/Rlp.sol	a3f36d268d2705e0b81407fa703b7a763b3c7aeb
contracts/messageService/lib/SparseMerkleTreeVerifier.sol	6d9b80f1ab1b0a0359104c14c60cf3a5bf5e48bf
contracts/messageService/lib/TimeLock.sol	77a43771c6c3babade08e6dc1935190e8686e961
contracts/messageService/lib/TransactionDecoder.sol	e9fbcde3e0653fd04f57e0326a90e547ade53ee8
contracts/token/LineaVoyageXP.sol	d97869c7c7f0bde8adb94dae5a145295d97cf19b
contracts/tokenBridge/BridgedToken.sol	7a4f73f0acb2a3c21f3e1bd79fdf9897b241bd2b
contracts/tokenBridge/CustomBridgedToken.sol	7c856bbc41696b45dfb571c84984693818c1682a
contracts/tokenBridge/TokenBridge.sol	16a363e129a46c22b1765bfc36f91103343cf1a8
contracts/tokenBridge/interfaces/ITokenBridge.sol	5fb239e774dfa43f4faf5ab7a968d129f0cc13a9
contracts/verifiers/PlonkVerifier.sol	8a9a131e8f37bca19c7cf496b4959468a66798b7
contracts/verifiers/PlonkVerifierFull.sol	0848f17e46a807d47131d0b720ee2aa7452bafe9
contracts/verifiers/PlonkVerifierFullLarge.sol	a9432b777f28e4ed4c11f8770b733d2ab51a21d9
contracts/verifiers/Utils.sol	3ee81fcc5b7891f687ba19598e6920317d90b06a

Appendix 2 - Disclosure

Consensys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.