AiLayer Labs 6079 Smart Contracts

Security Assessment (Summary Report)

June 14, 2024

Prepared for:
Ivan Ravlich
AiLayer Labs

Prepared by: **Michael Colburn, Alexander Remie, Priyanka Bose, and Suha Hussain**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineering directors were associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

Michael Brown, Head of Artificial Intelligence / Machine Learning
michael.brown@trailofbits.com

The following consultants were associated with this project:

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Priyanka Bose, Consultant
priyanka.bose@trailofbits.com

Suha Hussain, Consultant
suha.hussain@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
May 2, 2024	Pre-project kickoff call
May 14, 2024	Delivery of report draft
May 14, 2024	Report readout meeting
June 14, 2024	Delivery of summary report

Project Targets

The engagement involved a review and testing of the following target.

6079-contracts

Repository	https://github.com/Ai-Layer-Labs/6079-contracts
Version	`03300ae4d56859522890028e58f6ac1785e8a029`
Type	Solidity
Platform	EVM

Executive Summary

Engagement Overview

AiLayer Labs engaged Trail of Bits to review the security of its 6079 smart contracts at commit `03300ae`. These contracts facilitate the allocation of a reward token to users who stake Lido's stETH in the system on Ethereum; they also handle the cross-chain communication necessary to actually mint the accrued rewards on Arbitrum.

A team of four consultants conducted the review from May 6 to May 14, 2024, for a total of three engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the codebase, using automated and manual processes. We also conducted a lightweight design review based on the 6079 Proof of Inference Protocol (PoIP) white paper.

Observations and Impact

Our review of the `Splitter`, `LinearDistributionIntervalDecrease`, `RewardClaimer`, and `Distribution` contracts focused on identifying any issues that could affect the calculation of token shares over time to the different recipient groups, disrupt the accuracy of stETH stake accounting, prevent the distribution of rewards to stakers, or introduce errors in bookkeeping when updating the pools over time. We also reviewed the `L1Sender`, `L2MessageReceiver`, `L2TokenReceiver`, `Token`, `RewardClaimer`, and `Distribution` contracts for issues that could prevent minting or token-bridging operations from being transmitted correctly, allow a user to transfer or mint more tokens than expected, or result in improper integration with the Arbitrum, LayerZero, or Uniswap contracts.

The contracts included in the `@layerzerolabs` directory were considered out of scope for this review. The AiLayer Labs team also acknowledged the possibility of a negative stETH rebase, so we did not prioritize further exploration of this scenario.

We identified three issues during our review. One medium-severity issue and one informational-severity issue are the result of missing input validation of function parameters, which could at worst cause funds to erroneously be burned or at least lead to a confusing user experience. The remaining issue is of low severity and concerns important state changes that omit on-chain events, which could make tracking the system off-chain more difficult.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that AiLayer Labs take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Continue to improve the documentation, especially of lower-level implementation details.** Adding more inline comments to the code, especially in contracts such as `Splitter` and `Distribution`, will help make the overall system easier to follow.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The contracts use a modern compiler version that includes built-in overflow protection. The more complex math has been abstracted out to a stateless library, which facilitates testing. Consistent use of a scaling factor helps to minimize errors due to loss of precision.	**Satisfactory**
Auditing	Most state-changing functions emit events, though we did identify several functions that may benefit from additional event coverage (TOB-6079-2). We were not provided with an incident response plan.	**Moderate**
Authentication / Access Controls	There are several explicit roles within the system: each contract's owner, the fee owner, and the token minters. The owner capabilities for the `Splitter` and `Distribution` contracts are documented, but the remaining contracts and roles would benefit from explicit documentation as well.	**Satisfactory**
Complexity Management	Overall, the contracts are broken up into logical pieces. In some areas, the intended functionality is difficult to follow due to the minimal inline comments and the fact that much of the contract state is tracked in structs that are defined only in the interface contracts, meaning they require frequent cross-referencing. Additionally, the way rates are used in some of the contracts is somewhat unintuitive, as they seem to be closer to indexes than rates, and both the `Splitter` and `Distribution` contracts have pools that behave slightly differently, which makes the overall system even more difficult to understand.	**Moderate**
Decentralization	With the exception of the token contract, all of the	**Moderate**

	contracts in this system are intended to be upgradeable (though upgradeability can be disabled in the `Splitter` and `Distribution` contracts). The token contract's owner controls the minter and determines who can make changes to the LayerZero OApp configuration. The `Splitter` and `Distribution` contract owners can unilaterally change the rate of future rewards without notice. Documentation suggests that the ownership/governance structure is still a work in progress and may initially be held by a multisignature wallet controlled by the foundation, though the configuration of this wallet is unclear. Under normal operating conditions, the rewards are allocated and minted programmatically.	
Documentation	The high-level design and developer documentation adequately describe the intended functionality of the system, though there are some gaps around lower-level implementation details. The contracts themselves have fairly minimal inline comments. NatSpec comments are limited to externally exposed functions and are in the contract interfaces. Increasing coverage especially of internal functions will help make the codebase's expected functionality easier to understand.	**Moderate**
Low-Level Manipulation	The contracts do not use any low-level calls directly. Assembly is used only to unpack the sender address in the `L2MessageReceiver` contract.	**Satisfactory**
Testing and Verification	The contracts have adequate test coverage of positive and negative cases, though they generally rely on hard-coded values. Extending the tests to support a wider range of values and to leverage techniques such as fuzzing where possible would be beneficial. The test suite also includes some fork tests, and the system documentation includes instructions for manually testing the system on testnets.	**Satisfactory**
Transaction Ordering	We did not identify any transaction ordering vectors that could maliciously affect user funds or the system as a whole.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of zero-value checks in setter functions	Data Validation	**Informational**
2	Lack of event generation	Auditing and Logging	**Low**
3	Risk of token loss due to lack of zero-value check of destination address	Data Validation	**Medium**

Detailed Findings

1. Lack of zero-value checks in setter functions	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-6079-1
Target: Various files	

Description

Certain functions fail to validate incoming arguments, so callers of these functions could mistakenly set important state variables to a zero value, misconfiguring the system.

For example, the `Distribution_init` function in the `Distribution` contract sets the `depositToken`, `l1Sender`, and `splitter` variables to the addresses passed as arguments without checking whether any of the values are the zero address. This may result in undefined behavior in the system.

```
function Distribution_init(
    address depositToken_,
    address l1Sender_,
    address splitter_,
    FeeData calldata feeData_,
    Pool calldata pool_
) external initializer {
    __Ownable_init();
    __UUPSUpgradeable_init();

    depositToken = depositToken_;
    l1Sender = l1Sender_;
    splitter = splitter_;

    ...
}
```

Figure 1.1: The `Distribution_init` function (`Distribution` contract, L38–54)

In addition to the above, the following setter functions also lack zero-value checks:

- `Distribution`
    - `editPool`

- `L1Sender`
    - `setRewardTokenConfig`
- `L2MessageReceiver`
    - `setParams`
- `RewardClaimer`
    - `RewardClaimer__init`
    - `setSplitter`
    - `setL1Sender`

Exploit Scenario

Alice deploys a new version of the `Distribution` contract. When she invokes `Distribution_init` to set the contract's parameters, she mistakenly enters a zero value, thereby misconfiguring the system.

Recommendations

Short term, add zero-value checks to all function arguments to ensure that callers cannot set incorrect values and misconfigure the system.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts.

2. Lack of event generation	
Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-6079-2
Target: Various files	

Description

Multiple user operations do not emit events. As a result, it will be difficult to review the contracts' behavior for correctness once they have been deployed.

Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions; malfunctioning contracts and attacks could go undetected.

The following operations should trigger events:

- `Splitter`
  - `Splitter__init`
  - `updatePool`
  - `updateGroupShares`
  - `removeUpgradeability`
- `L1Sender`
  - `L1Sender__init`
  - `setRewardTokenConfig`
  - `setDepositTokenConfig`
  - `updateAllowedAddresses`
  - `sendDepositToken`
  - `sendMintMessage`
- `Distribution`

    - `removeUpgradeability`
- `L2MessageReceiver`
    - `setParams`
- `L2TokenReceiver`
    - `editParams`
    - `withdrawToken`
    - `withdrawTokenId`
- `RewardClaimer`
    - `RewardClaimer__init`
    - `setSplitter`
    - `setL1Sender`
- `Token`
    - `updateMinter`

Exploit Scenario
An attacker discovers a vulnerability in any of these contracts and modifies its execution. Because these actions generate no events, the behavior goes unnoticed until there is follow-on damage, such as financial loss.

Recommendations
Short term, add events for all operations that could contribute to a higher level of monitoring and alerting.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

3. Risk of token loss due to lack of zero-value check of destination address	
Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-6079-3
Target: Various files	

Description

We identified two functions that fail to verify whether the destination address is the zero address before sending tokens. This can lead to the unintentional burning of tokens.

First, as shown in figure 3.1, the `claim` function in the `Distribution` contract takes a `receiver` address, which is user-provided, and subsequently uses it to send native tokens via the `sendMintMessage` function in the `L1Sender` contract. Consequently, if the user mistakenly provides the zero address as the `receiver` argument, the tokens will inadvertently be burnt.

```
function claim(address receiver_) external payable {
    address user_ = _msgSender();
    UserData storage userData = usersData[user_];
    ...
    // Transfer rewards
    IL1Sender(l1Sender).sendMintMessage{value: msg.value}(receiver_,
    pendingRewards_, user_);
    emit UserClaimed(user_, receiver_, pendingRewards_);
}
```

Figure 3.1: The `claim` function (`Distribution` contract, L184–209)

Second, the `claim` function in the `RewardClaimer` contract does not verify whether the `receiver_` argument is the zero address; tokens will also be burned if a user supplies the zero address for this argument.

Exploit Scenario

Alice invokes the `claim` function in the `Distribution` contract to claim reward tokens. However, by mistake she supplies the zero address as the receiver address. Consequently, the claimed rewards are transmitted to the zero address, causing the tokens to be irreversibly lost.

Recommendations

Short term, add zero-address checks to the function arguments to ensure that callers cannot set incorrect values that result in the loss of tokens.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's CI pipeline, pre-commit hooks, or build scripts.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendation is not associated with any specific vulnerabilities. However, it will enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Update the name of the `Distribution_init` function to match the other initialization functions.** Most of the upgradeable contracts in the repository follow the pattern `ContractName__init` for naming the initialization function, but the `Distribution` contract's initialization function uses only one underscore instead of two.

D. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On June 4, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the AiLayer Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the three issues described in this report, AiLayer Labs has resolved one issue and has partially resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Lack of zero-value checks in setter functions	**Partially Resolved**
2	Lack of event generation	**Resolved**
3	Risk of token loss due to lack of zero-value check of destination address	**Partially Resolved**

Detailed Fix Review Results

TOB-6079-1: Lack of zero-value checks in setter functions
Partially resolved in PR #9. Input validation that checks for default values was added to the `Distribution_init` function in the `Distribution` contract and to the functions in the `RewardClaimer` contract. The remaining functions highlighted in the issue were unmodified. The AiLayer Labs team accepts the risk of omitting input validation from these `onlyOwner` functions.

TOB-6079-2: Lack of event generation
Resolved in PR #9. All of the functions highlighted in this finding now emit events to reflect any changes to the system's state.

TOB-6079-3: Risk of token loss due to lack of zero-value check of destination address
Partially resolved in PR #9. Input validation was added to the `claim` function in the `Distribution` contract to ensure the mint message will not attempt to include the zero address as the receiver. No changes were made to the `claim` function in the `RewardClaimer` contract. The AiLayer Labs team accepts the risk of omitting input validation from these `onlyOwner` functions.

E. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

F. Lightweight Design Review

We performed a best-effort lightweight design review of the 6079 Proof of Inference Protocol (PoIP), the overarching protocol for this system. We based our review on the 6079 PoIP white paper, focusing on identifying design flaws that could result in a compromise of confidentiality, integrity, or availability. This section provides our recommendations for improving the PoIP specification and corresponding system implementation. Note that any changes or deviations from the white paper will impact and potentially invalidate the findings of this review.

Overview

This section details potential attack vectors and implementation issues in PoIP systems and provides recommendations for further specification of PoIP and for the secure implementation of such systems. Specifically, we find that PoIP can be compromised using correctness issues in the implementation and management of transactions, the insecurity of GPUs and the GPU ecosystem, vulnerabilities in the inference engine, and protocol-level issues stemming from cryptographic and ML system properties.

Our top-level recommendations are as follows:

1. Assess the correctness of the implementation and management of transactions, especially in proving transaction inputs across nodes.
2. Investigate and develop appropriate mitigations and defenses for security issues involving GPUs and the GPU ecosystem.
3. Construct a secure-by-default inference engine standard.
4. Further specify and analyze the protocol for cryptographic and ML system issues.

Background

According to the white paper, “PoIP leverages a cryptoeconomic security approach to incentivize desired behavior and penalize malicious actors.” While it is imperative that this cryptoeconomic security approach be sound, both as specified and as implemented, it is also important to determine whether the incentives specified in the protocol are the only incentives that exist for actors interacting with the system in practice. Any downstream benefit of exploitation will undermine the guarantees and overall security of PoIP.

To that end, this review focused on two major risks for PoIP systems:

1. The ML development stack, especially the GPU ecosystem, contains numerous components that could have unidentified security vulnerabilities that have yet to undergo thorough evaluation by security professionals.

2. The inclusion of ML models into cryptographic protocols and other systems can lead to emergent exploitable vulnerabilities that combine model and system security issues (such as NeuralHash collisions in ImageNet).

Implementing and Managing Transactions

Transactions are fundamental to PoIP. Exploitable vulnerabilities in their implementation and management would greatly impact the PoIP system's overall security and result in financial losses.

Recommendation: Thoroughly assess the correctness of transaction management and implementation, especially with regard to proving transaction inputs.

Transaction operations are centered around a distributed hash table (DHT) shared by all the nodes. These operations involve synchronizing between and routing information to and from DHTs throughout the network. Take the following steps to secure transaction operations:

1. Ascertain whether there should be controls on flagging transactions to prevent too many transactions from being flagged and information from being transferred to long-term storage on chain.

2. Evaluate whether an attacker can manipulate the synchronization of DHTs or place incorrect or manipulated transactions onto the central DHT.

3. Confirm that denial of service of or rerouting from the Probabilistic Proof Layer, which could result in divergences between the central DHT and the system's nodes, is not possible.

In PoIP, a process reliant on Merkle trees is used to ensure the correctness of transaction inputs, specifically the data sent to each node in the system. Manipulating this process permits an attacker to send incorrect data to the nodes in the system and undermine the system's integrity. Ensure that this component holds the following properties:

1. The Merkle tree implementation should not be vulnerable to second-preimage attacks, in which attackers create input data other than the original with the same Merkle hash root. Malicious actors may be able to chain such attacks with model evasion.

2. Order should be preserved to minimize the impact of adversarial input manipulation. Note that if the model is online, an attacker may be able to reorder data points to conduct model poisoning attacks (Shumailov et al.).

3. Subtask computation must be correct across a wide variety of tasks. Determine whether it is possible for operators to be reordered to deliver a backdoored model (Goldwasser et al.).

4. Input preprocessing, hash construction, and Merkle tree reconstruction must be correct across a wide variety of inputs.

5. The Merkle tree implementation should make a distinction between the inner and leaf nodes. Otherwise, attackers may be able to provide incorrect proofs of transaction, a vulnerability previously discovered in Bitcoin.

6. GPU side channels and batching variations should have minimal impact on correctness.

Managing the Insecurity of GPUs

Since the verifiable, secure coordination of GPUs is a primary objective of this protocol, the impact of GPU and GPU ecosystem security on PoIP systems warrants further investigation.

Recommendation: Review and specify authorized GPU hardware variations and configurations (and determine whether multitenant systems should be authorized) and the incorporation of GPU tooling. Implement the appropriate mitigations and defenses in the GPU server standard.

The white paper is unclear as to whether each node maps to a distinct and separate GPU device, putting the protocol at risk in multitenant implementations (i.e., developers building the protocol based on an unclear specification could introduce unsafe configurations). Moreover, GPUs differ in architecture, memory limits, and memory; depending on these factors, certain GPUs may impact components such as aggregation. Components of the GPU ecosystem may also have unintended interactions with other system components such as wallets.

Due to GPU design limitations, new confidentiality, integrity, and availability risks exist when a GPU is shared across users or processes, as in so-called multitenant systems. For confidentiality, disclosure vulnerabilities can result in side channels and covert channels. For availability, a malicious actor could make a GPU node in PoIP slow down, hoard available memory against, or crash the processes of other GPU nodes. For integrity, persistence vulnerabilities can lead to insecure environments for subsequent users.

To adequately mitigate these risks (e.g., using memory scrubbing), it is critical to specify authorized GPU configurations in the protocol. The protocol should also specify verification methods for authorized configurations to mitigate or defend against Sybil attacks and the reintroduction of decommissioned nodes, which can inherit the vulnerabilities of sequential multitenant systems.

Recommendation: Explore the risks associated with side channels and covert channels. Integrate relevant mitigations and defenses into the GPU server standard.

In the current PoIP version, a dishonest or malicious GPU node can impact confidentiality, load balancing, swarm optimization, and, most notably, token distribution. To do so, side channel attacks can be leveraged, undermining the game-theoretic guarantees of PoIP.

Side-channel and covert-channel vulnerabilities—which include information leakage due to incorrect implementations, as in the GPU vulnerability known as LeftoverLocals—can violate the implicit isolation between GPU nodes in PoIP. Specifically, if nodes can eavesdrop on one another, they could try to appropriate the results of computation performed by other nodes. An attacker would thus be able to increase their reputation and control over the network. The viability of this integrity attack is reinforced by the ability to stake tokens and coordinate using the DHT, especially if multiple malicious GPUs collaborate. The diagnostic tasks performed on GPU nodes in the Probabilistic Proof Layer should be designed to withstand leakage and malicious GPU nodes.

Securing the Inference Engine Standard

PoIP details a standard for the inference engine, which is intended to be open-source software that handles deployment of ML models and execution of computations. Ensure that the inference engine standard is secure by default. This will help protect both the participants in the protocol and developers building on top of the protocol.

Recommendation: Rewrite the system to treat the agent and gateway nodes as potentially malicious, and set sufficient constraints to prevent/mitigate such risks.

These constraints should be further described in the specification or added to the inference engine standard. Additionally, we recommend exploring whether there should be a stronger separation of concerns between actors, as both the agent and gateway nodes have many responsibilities and capabilities. We also recommend further exploring proper constraints for AI agents, such as not providing them unbounded access to external resources. Consider the following scenarios:

1. An agent uses an energy-latency attack to force a denial of service of the model.
2. Prompt injection forces an AI agent to drain a wallet or forge transactions.
3. A malicious actor uses a gateway node to manipulate the compilation of responses and flag compliant nodes as noncompliant.

Recommendation: Ensure that the inference engine standard appropriately constructs the inference task and follows ML security best practices.

The construction of the inference task is critical to the success of PoIP. Therefore, we recommend the following actions:

1. Ensure that the inference task is strictly specified and not easy to fake.

2. Investigate how attacks on PoIP systems can change based on the inference task. In some domains, it can be assumed that the attacker has access to the training data, which can aid exploitation (Zhang et al.).

To minimize both supply-chain and model risks to application developers and protocol participants, the inference engine standard should adhere to the following guidelines:

1. Enforce the use of secure, authorized model file formats such as safetensors. (See our security review of the safetensors library.) Insecure file formats such as pickle can help an attacker tamper with model weights, opening the door for model vulnerabilities such as backdoors. Consider an attacker that can inject a backdoor that sabotages nodes by forcing them to return incorrect results.

2. Implement proper constraints on model capabilities, especially to minimize the impact of prompt injection and privacy concerns. Refer to NVIDIA's guidelines for more information.

3. Minimize the attack surface introduced by delivering models as serverless microservices.

4. Audit and vet dependencies.

5. Control the model life cycle by implementing proper versioning and monitoring.

6. Include model cards with detailed information on the definition and selection of requirements, dependencies, and the underlying stack.

7. Institute proper controls over data ingestion by models.

Conducting Protocol Analysis

We recommend further analyzing the protocol for any underlying issues related to cryptography and machine learning and discussing any findings resulting from this analysis in the specification.

1. Conduct a thorough design review and threat model of PoIP. Prioritize determining whether the separation of concerns between actors is sufficient and whether there are unspecified incentives for attackers that can violate the cryptoeconomic guarantees of PoIP.

2. Probe the interplay between potential vulnerabilities in the protocol, infrastructure, and models, given that infrastructure emissions are sent to both the GPU and gateway nodes.

3. Explore how decentralized PoIP is in practice. The ML life cycle and stack is very centralized in compute providers and datasets (Carlini et al.). Consult the Trail of Bits white paper on unintended centralities in blockchains, written by Sultanik et al., for a variety of metrics to measure the true decentralization of a blockchain protocol.

4. Investigate the system's confidentiality, specifically whether PoIP violates intrinsic model privacy (Debenedetti et al., Pasquini et al.) and how information such as metadata should be securely and appropriately transmitted. (See this white paper for a discussion of the risks of insecure metadata transmission in Zcash.)