The Phylax Credible Layer

Odysseus Lamtzidis et al., Phylax Systems

October 2024

Abstract

We introduce the Phylax Credible Layer, a novel blockchain security mechanism designed to prevent and mitigate hacks in decentralized applications by integrating at the network's base layer. The protocol comprises two core mechanisms built upon a single primitive: Assertions. These are out-of-band computations over the base layer's state, associated with specific smart contracts, enabled at any time, and computed off-chain, allowing for a broader range of constraints than possible on-chain. The mechanisms enable dApps and the base layer to coordinate on valid and invalid states:

1. The State Oracle (SO): A mechanism where assertions are defined and associated with specific smart contracts (Assertion Adopters). Assertion Consumers (Users) can submit proofs about transactions that invalidate an Assertion, either as Proof of Possibility (potential future transaction) or Proof of Realization (on-chain executed transaction). Proofs are submitted only when an assertion is invalidated.
2. The Assertion Enforcement Mechanism: Actors (Assertion Enforcers) bond and commit to enforcing that transactions they submit do not invalidate any Assertion. These can be any entity with transaction inclusion veto power, such as block-builders, sequencers, or searchers. They receive fees for this service and are slashed if a Proof of Realization is submitted for any state resulting from their transactions.

This work explores the mechanism's design space, security model, and use-cases. It discusses the implementation of the first iteration, Ajax, designed to integrate with OP-Stack-based rollups. It also examines potential fee structures and token incentive designs to align various Credible Layer actors. Finally, it briefly explores how integrating the Credible Layer with a base layer might affect the network's Fork-Choice Rule or State Transition Function, enabling "Contract-based Soft-Forking".

This is by no means a complete work or peer-reviewed paper, but rather a first exploration into the design space of the Credible Layer and it's primitives. There are many open questions that we have not addressed, such as the formalization of the security model and the crypto-economic design. We hope that this work will serve as a foundation for future research into the Credible Layer, as we progressively move towards a production-ready implementation.

1 Current Approaches to Hack Prevention

The proliferation of decentralized finance (DeFi) has led to an increase in sophisticated attacks on blockchain protocols. This section examines current methodologies for preventing and mitigating such attacks, highlighting their strengths and limitations.

1.1 Front-running and Next-block Mitigation

With the rise of Maximum Extractable Value (MEV) [13], the industry has witnessed the emergence of "generalized frontrunners" [28] - automated systems that simulate transactions to identify profitable opportunities. Inadvertently, these systems began to frontrun malicious transactions, leading to the development of frontrunning services, such as Phalcon [39], as a hack prevention technique.

1.1.1 Limitations of Frontrunning

While frontrunning can potentially mitigate some attacks, it faces significant challenges:

- **Private Mempools:** Attackers can utilize private transaction pools, such as Flashbots [19], or setting up their own validator, to circumvent frontrunning attempts.
- **Dark Mempools:** In the event where well established Private Mempools start collaborating with hack prevention services, we expect the emergence of "Dark Mempools". Private Mempools associated with validators in permissive jurisdictions which will offer transaction inclusion for hackers that are driven out of the other private mempool services.

1.1.2 Next-block Mitigation

Next-block mitigation strategies aim to pause or alter protocol operations in the block immediately following a detected exploit. While this approach could have been used successfully in cases such as the Euler [17] and Nomad [36] hacks, it relies on a core assumption:

The strategy assumes that attacks span multiple blocks, which is often due to inefficient execution rather than technical necessity. While hackers may be extremely sophisticated in finding vulnerabilities, they frequently botch the execution of their attacks. This highlights a discrepancy between the sophistication of vulnerability discovery and attack execution.

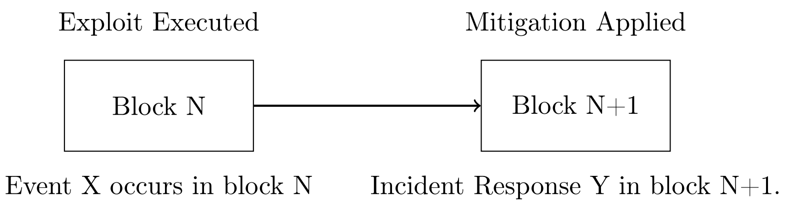

Figure 1: Next-block Mitigation Strategy

Both frontrunning and next-block mitigation operate at the application level, which is fundamentally insufficient for comprehensive hack prevention. A more robust approach necessitates integration at the base layer of the blockchain infrastructure.

1.2 Heuristic Analysis of Deployed Smart Contracts

Another prevalent approach involves the use of various automated testing and analysis techniques [29] to identify vulnerabilities in deployed smart contracts. These techniques include:

1. Fuzzing [23]: An automated testing method that generates random inputs to test for vulnerabilities.
2. Formal Verification and Symbolic Testing [48]: A technique that uses symbolic execution to explore multiple execution paths and identify potential vulnerabilities. It uses mathematical methods to prove or disprove the correctness of smart contract code with respect to certain specifications.
3. Static Analysis [47]: Examining the source code or bytecode without executing it to identify potential security issues.

These techniques aim to comprehensively assess smart contracts for potential vulnerabilities before and after deployment.

1.2.1 Advantages and Limitations

While these techniques can detect well-known vulnerability patterns (e.g., reentrancy), they face several challenges:

- **Development Overhead:** The effort required to develop and maintain these solutions often exceeds that of creating platforms enabling developers to define protocol-specific security parameters.
- **Scalability Issues:** Increased computational complexity for higher accuracy leads to scalability challenges in real-time transaction and contract scanning.

- **Reliability Concerns:** These techniques are inherently probabilistic in nature and may result in false positives or missed vulnerabilities. While this approach may uncover common vulnerabilities, it simply can't be used for real-time hack prevention, as it would block too many users.

While heuristic analysis has shown promise, particularly in projects like Fuzz.land [23], we believe that it's a complimentary tool that should be used by developers prior to the deployment of their smart contracts.

The computational resources required to run these tools and the risk of false positives mean that it's almost untenable to use them for real-time hack prevention.

1.3 The Need for a Paradigm Shift

The limitations of current hack prevention methodologies underscore the need for a fundamental shift in approach. Effective security measures must:

- Operate at the base layer of blockchain infrastructure
- Provide better security guarantees than the ones provided by the current approaches
- Enable protocol developers to define and enforce security parameters specific to their applications
- Scale efficiently to accommodate the growing complexity of decentralized ecosystems

The following sections introduce the Phylax Credible Layer, a novel security mechanism designed to address these challenges and provide robust protection against sophisticated attacks in decentralized systems.

2 The Credible Layer

In this section we describe the underlying mechanism of the Credible Layer. We first briefly talk about the cornerstone primitive that serves as the kernel of the mechanism, then the mechanism(s) it spawns and finally how these can be used to build a protocol for the prevention of hacks.

2.1 Assertions

The cornerstone primitive of the system is what we call *Assertions.* An assertion A is a function that maps a blockchain state S to a boolean value:

$$A : S \to \{\text{true}, \text{false}\} \tag{1}$$

Where:

- S represents the state of one network
- `true` indicates a valid state
- `false` indicates an invalid state

It is worth mentioning that an assertion could also be defined as a function over multiple blockchain states, i.e. $S_1, S_2, ..., S_n$ for n blockchain states. We refer to these as *Cross-Chain Assertions* but we will leave the exploration of this topic for future work.

Assertions are powerful primitives, as they are not executed on chain, but off chain and a proof of the computation result is submitted on chain. This enables the assertions to be arbitrarily complex and with encoded logic that is impossible to express on chain, such as evaluating state over multiple blocks.

2.1.1 Assertions as Intents

Assertions can be conceptualized as a form of negative intent, analogous to the recently popularized concept of intents in blockchain systems [27]. While traditional intents express desired outcomes, assertions define states that must be avoided. This paradigm shift from prescriptive to proscriptive specification offers a powerful abstraction for system security.

In the context of decentralized applications (dApps), assertions function as declarative safeguards against undesirable states. This approach is particularly efficacious for security implementations, as it circumvents the need to anticipate all possible attack vectors. Whether the threat originates from code vulnerabilities [54], compromised private keys [43], or even compiler-level exploits [51], a well-crafted assertion can prevent the system from entering an invalid state.

The assertion mechanism thus provides an accessible and robust tool for developers seeking to enhance their dApp's security posture. By defining what states must not be reached, rather than exhaustively enumerating all permissible states, assertions offer a more flexible and comprehensive security model. This approach ensures that, regardless of the specific exploit path an attacker might discover, the system remains protected as long as the invalid state is encompassed by the assertion.

2.1.2 Assertion Examples

In this section we will explore a few examples of assertions that could be used for a wide range of protocols. Broadly speaking, assertions can be more ideal to perform a wide range of checks for the following reasons:

1. They are not executed on chain, so they can encode arbitrarily complex logic, for a fraction of the cost.
2. They can be added and removed on chain, without needing to redeploy the contract. It enables a kind of dynamic security mechanism, but with the expressiveness of on-chain code.

3. They can only be used to limit functionality, not enable it. Modifying the on-chain code to enable a check can lead to the introduction of a new vulnerability, which is not the case with assertions.

We are still extremely early in defining the assertion standard, so we expect many more use-cases to be discovered, as we expand what is possible with assertions.

Example 1: Critical Variable Changes A simple assertion could be defined as check on a critical variable, such as the owner of a contract, an admin or the implementation address of a proxy.

`Variable A should never change.`

The benefit of expressing this as an assertion is that a vulnerability in the logic of the contract can not allow the attacker to make the change. If the team wants to change the owner, they will have to disable the assertion first and then make the change. It's an extra layer of security and friction to ensure that certain critical variables, that change rarely, are not changed maliciously. A recent example of this is the Radiant Capital hack [42].

Assertion 1

Variable A should never change from its initial value.

Example 2: Market Manipulation A more complicated assertion could be defined a check for market manipulation, which have resulted to numerous hacks such as the one in Cream Finance [11]. The most usual case for market manipulation is by manipulating the value of the oracle by leveraging flash loans (a type of on-chain loan that is executed and settled in a single transaction).

A few examples of assertions that could be useful for various protocols are the following:

Assertion 2

More than X% of the positions of the lending market shouldn't get underwater over Y blocks or Z transactions.

Assertion 3

The TVL of the lending market should never drop more than Y% over Z blocks or J transactions.

Assertion 4

The oracle value shouldn't change by more than X% over Y blocks or Z transactions.

Assertion 5

The transaction trace should never include a flashloan call.

Assertion 6

If the Oracle value hasn't been updated in the last X blocks, then the protocol shouldn't be interacted with.

Example 3: Ensure Intended Effect Another example of assertions are the ones that validate that the user transaction has the intended effect. It's an extra validation of the intended effect which may leverage data that is either too expensive to compute on-chain (e.g running average) or impossible (e.g over many blocks). In a way, developers can port their invariant tests as assertions which validate transactions.

Assertion 7

If the user deposits X amount of tokens in pool Y, then they should get Z amount of shares.

2.2 Credible Layer Fundamentals

An assertion, as demonstrated, can encode complex security requirements and invariants that would be impractical or impossible to enforce directly on-chain. From this foundational primitive, we construct two complementary mechanisms that work together to provide comprehensive security guarantees.

Let's define the actors of the system:

- **Assertion Adopters (AA)**: A smart contract on the network which has an assertion tied to it.
- **Assertion Submitters (AS)**: The entity that develops and provides assertions to AAs. Most likely security researchers, protocol engineers, etc.

- **Assertion Enforcers (AE)**: The entity that is able to enforce an assertion during the creation of a block. It is a block builder or a block proposer. Often both at the same time.
- **Assertion Consumers (AC)**: The entity that consumes whether the assertion holds true or consumes the assertion itself to produce proofs of (possible) invalidation. It can be any of the above actors, an end-user or even a smart contract.

Let the actors be grouped into two main groups:

1. Those who offer security: Assertion Enforcers, Assertion Submitters, Assertion Consumers
2. Those who need security: Assertion Adopters

Let the formation of two markets be defined as:

1. Assertion Market: A mechanism for giving incentives for the definition of assertions and the signaling of whether they have been invalidated or can be invalidated.
2. Assertion Enforcement Market: A mechanism for giving incentives to actors to enforce the assertions by not submitting transactions to the network which results to their invalidation.

And more specifically:

1. **Assertion Market**:
   - Assertion Submitters submit assertions to Assertion Adopters.
   - Assertion Adopters offer rewards to:
     - Assertion Consumers for producing proofs of (possible or realised) invalidation.
     - Assertion Submitters for providing assertions.
2. **Assertion Enforcement Market**:
   - Assertion Enforcers offer assurances to Assertion Adopters that the assertions will be enforced.
   - Assertion Adopters offer rewards to Assertion Enforcers for the assertion enforcement.

Thus, the Assertion Market is a market for the definition of assertions and the signaling of whether they have been invalidated or can be invalidated. The Assertion Enforcement Market is a market for the enforcement of assertions by not submitting transactions to the network which results to their invalidation. We see how the two markets are linked, as the Assertion Market is a prerequisite for the Assertion Enforcement Market. All of which is enabled by the primitive of the Assertion and the actors that participate in the markets.

The following are the definitions of the primitives that are used in the mechanism:

Definition 2.1 (Invalidating Transaction)**.** An Invalidating transaction is a valid transaction that once applied to the state of the Network via the State Transition Function, an assertion of an Assertion Adopter becomes invalidated.

Definition 2.2 (Proof of Possibility - PoP)**.** It is a zk proof that is submitted by the Assertion Consumer and it proves that an Invalidating transaction exists, but has not been submitted to the Network. It doesn't reveal the transaction itself, so that the vulnerability is not exploited by other users of the Network.

Definition 2.3 (Proof of Realisation - PoR)**.** It is a zk proof that is submitted by the Assertion Consumer and it proves that an Invalidating transaction has been included in a block and thus finalized on the Network.

Definition 2.4 (Assertion Bounty)**.** An Assertion Bounty acts as a proactive incentive, payable to Assertion Consumers in exchange for submitting a Proof of Possibility to Assertion Adopters, rather than exploiting a perceived vulnerability.

Definition 2.5 (Assertion Mitigation)**.** It is an EVM call that is defined by the Assertion Adopter and associated with an assertion. If a valid Proof of Possibility or Realization is submitted, it is marked as executable and can be executed by anyone. Most likely it will be used as a kill-switch to pause the dApp of the Assertion Adopter before it gets exploited.

2.2.1 Credible Layer Overview

Thus, the Credible Layer is actually two mechanisms that work in tandem to support a variety of use-cases, one of which being hack prevention. The Assertion Market functions as the basis for defining and invalidating assertions, while the Assertion Enforcement Market consumes the output of the Assertion Market and acts as a coordinator between the Assertion Adopters and the Assertion Enforcers. The output relates to the Assertions that should be enforced, as also whether they have been invalidated or not, which signals whether the Assertion Enforcers should be slashed or not. Figure 2 provides an overview of the two mechanisms and how they compose.

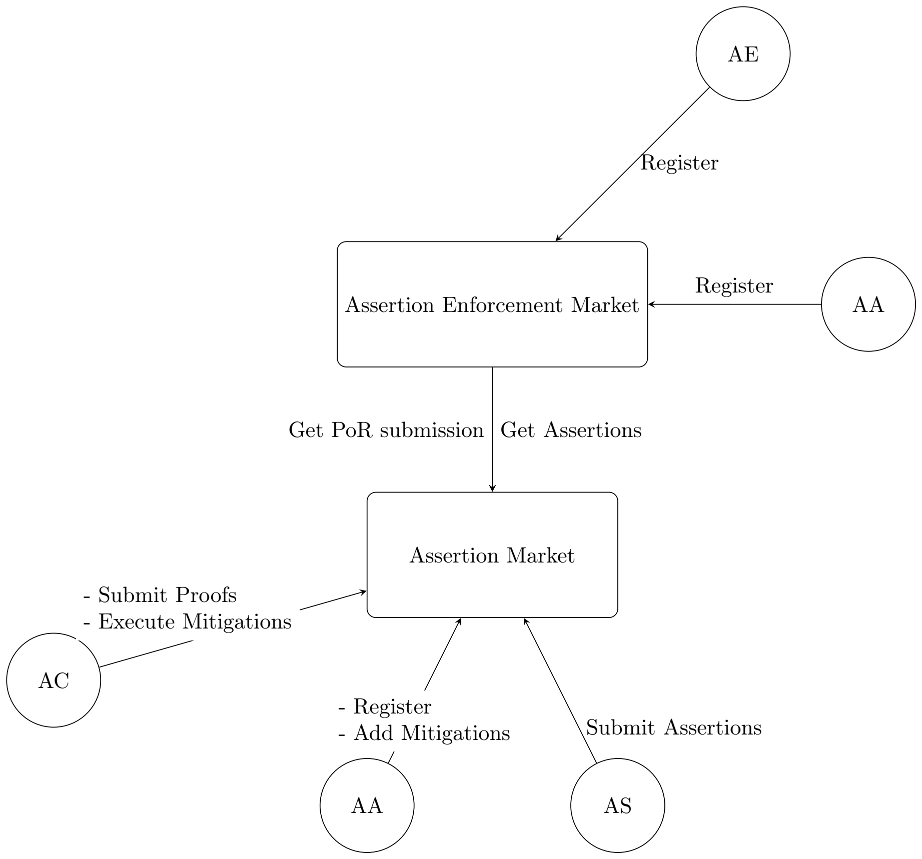

Figure 2: Assertion Market and Assertion Enforcement Market

2.3 Assertion Market - State Oracle (SO)

The Assertion Market or State Oracle (SO) is a on-chain mechanism which creates a marketplace between dApps who need Assertions and those who provide them. Moreover, it enables dApps to associate assertions with bounties or mitigations and, finally it serves as an oracle for whether the assertions have been invalidated or can be invalidated.

2.3.1 Assertion Market Flow

1. Assertion Adopters register with the protocol and deposit funds to their security budget.
2. Assertion Submitters register with the protocol and propose assertions to the Assertion Adopters.

3. Assertion Adopters identify and select assertions to whitelist. The Assertion Adopters post a bounty for the assertions, and add any relevant mitigations.
4. Assertion Submitters are rewarded for providing assertions and thus contributing to the security of the Assertion Adopters. Rewards can be claimed at any time, subtracting the amount from the security budget and adding it to that of the Assertion Submitters.
5. An Assertion Consumer submits a Proof of Possibility to the protocol, claiming the associated bounty for proving that a particular assertion may be invalidated.
6. The assertion is marked as *Possibly Invalidated* and the associated mitigation can be executed by anyone.
7. An Assertion Consumer submits a Proof of Realization to the protocol, without receiving bounty.
8. The assertion is marked as *Invalidated* and the associated mitigation can be executed by anyone.

It incentivises two behaviors:

1. Security researchers develop and publish assertions to the Assertion Market. Researchers will be rewarded whenever their assertion is added to Assertion Adopters' set of active assertions.
2. Convert blackhat hackers into whitehats, who may claim bounties, in exchange for identifying vulnerabilities, without requiring KYC.

One of the most important properties of this construct is that the submitter of the Proof of Possibility doesn't have to argument on whether the assertion has been invalidated or not. If they submit a Proof of Possibility, and there is a locked bounty by the Assertion Adopter, they can claim the bounty according to the rules set by the Assertion Adopter. This is in stark contrast with the current state of the art, where vulnerability submitters have to often argue with the team on whether the vulnerability lies within the scope of the bug bounty or not, as also have to go through KYC.

Figure 3 provides an overview of the flow of the Assertion Market and the submission of Proofs of Possibility. The flow of the Proof of Realization is omitted for brevity, but is nearly identical.

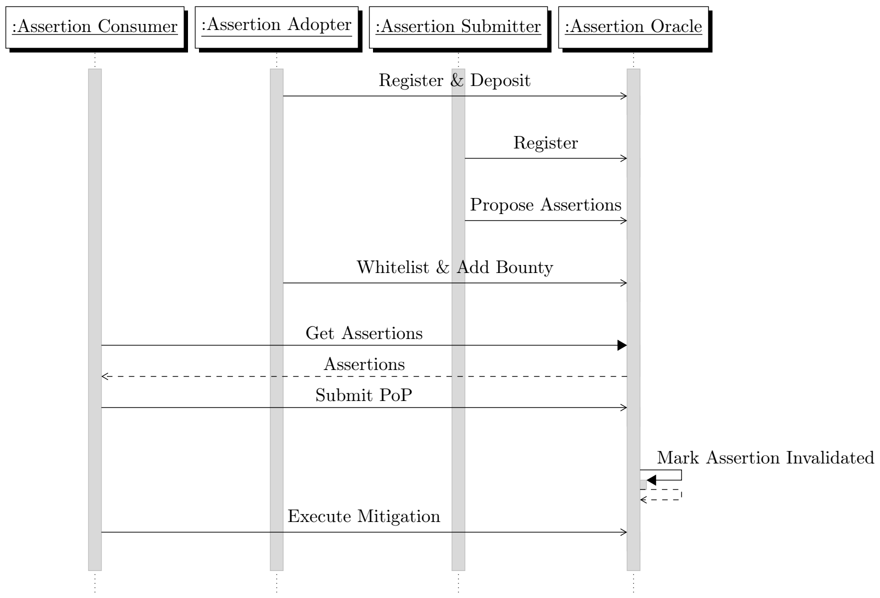

Figure 3: Assertion Definition and Proof of Possibility submission flow

2.3.2 Use cases

We believe the State Oracle will have wide-ranging implications for the industry and extend far beyond hack prevention and mitigation, however we have identified a handful of initial use-cases we would like to work with researchers to bring to market as part of our MVP.

Example 1: On-chain bounties The State Oracle's primary advantage lies in its deterministic nature. Assertions yield binary outcomes, eliminating the need for intermediaries to adjudicate validity. This enables protocols to formulate bug bounty programs as assertions, allowing security researchers to claim rewards by submitting Proofs of Possibility. This process mitigates subjective judgment and protracted disputes. Bounties can be assertion-specific or pooled for capital efficiency.

This framework incentivizes broader participation from security researchers by removing reliance on centralized entities for bounty disbursement.

Moreover, it encourages blackhats[1] to submit Proofs of Possibility, as they can claim bounties for invalidating assertions without direct exploitation. This incentive structure promotes a transition to whitehat[2] behavior, fostering vulnerability discovery rather than exploitation. The absence of KYC requirements preserves hacker anonymity.

[1]A blackhat hacker identifies and exploits protocol vulnerabilities.

[2]A whitehat hacker identifies vulnerabilities without exploitation, typically reporting findings to the protocol, sometimes for compensation.

Example 2: Escape Hatches Account Abstraction [15] enables the delegation of user actions to smart contracts, allowing users to define automated position unwinding in DeFi protocols without relying on intermediaries. This functionality can be leveraged to trigger automatic position liquidation upon the submission of a valid Proof of Possibility or Realization to the State Oracle for any Phylax-secured DeFi protocol.

Example 3: Automated Mitigation The State Oracle mechanism facilitates the definition of assertions that trigger on-chain actions to mitigate vulnerabilities in decentralized applications. This approach incentivizes both white-hats to provide Proofs of Possibility and protocols to define such assertions, enabling timely reactions to potential exploits before they materialize.

Example 4: Insurance Proof of Realization introduces a novel on-chain primitive for smart contracts to consume information about protocol hacks, enabling the development of automated on-chain insurance mechanisms. This concept, explored further in the whitepaper, is envisioned as an extension of the Credible Layer.

Example 5: Hack Prevention The most valuable use-case is the preemptive prevention of potential hacks. The State Oracle enables programmatic penalization of malicious actors, unlocking a new design space for security mechanisms, which we explore in subsequent sections.

Example 6: Arbitrary State Oracle While our discussion has primarily focused on security applications, the State Oracle's utility extends beyond this domain. Assertion Adopters can define non-security related assertions encoding undesirable state transitions, allowing Assertion Consumers to signal violations or build on top of the State Oracle's output, as a primitive.

2.4 Enforcement Market - The Credible Layer

The Credible Layer introduces a secondary market of Assertion Enforcers, incentivized to prevent the realization of state transitions defined in the State Oracle. It functions as a **Coordination Layer** between decentralized applications (Assertion Adopters) and the base layer (Assertion Enforcers), facilitating agreement on prohibited state transitions to prevent hacks.

The following section delves into the mechanism's details and the primitives employed to achieve the desired security properties.

2.4.1 A Hack in the Credible Layer

A hack is loosely defined as a state of the dApp deemed valid under the protocol's rules, but violates the intended business logic of the protocol.

In the context of the Credible Layer, **a hack is defined as a state of the dApp where some of it's assertions are invalidated**.

Essentially, we are making an attempt in defining something very arbitrary, in a clear and unambiguous way. This is a necessary step if we want to build a crypto-economic mechanism which penalizes actors in the case of a hack.

Likewise, it's useful to define what we mean by Assertion Accuracy and its impact on security vulnerabilities. This analysis is key for developing robust incentive mechanisms, especially in fee structures. Appendix A.1 provides a detailed examination of how assertion accuracy relates to the Credible Layer's overall security model.

2.5 Assertion Enforcement Mechanism

Let's consider an environment with a single Assertion Enforcer, such as an L2 with a centralized sequencer. Furthermore, let's assume that the Enforcer agrees to enforce any assertions that are defined by the State Oracle.

While this scenario simplifies our initial analysis, it's important to note that this assumption doesn't limit the generality of the mechanism.

The Credible Layer is designed to accommodate multiple Assertion Enforcers and more complex enforcement agreements, making it adaptable to various blockchain architectures and security models.

An overview of the mechanism flow can be described as follows:

1. The Assertion Enforcer signs up to the Credible Layer and deposits funds for it's security budget

2. The Assertion Adopter signs up to to the Credible Layer and deposits funds for it's security budget

3. The Assertion Adopter submits assertions which get activated, these are submitted to the State Oracle

4. The Assertion Enforcer ensures that all new blocks do not include an Invalidating transaction

5. The Assertion Enforcer claims rewards from all the registered Assertion Adopters, which get subtracted from their security budget and added to the security budget of the Assertion Enforcer

6. If a valid PoR is submitted, then the Assertion Enforcer is slashed and funds are distributed to the Assertion Adopter for which the PoR was submitted

Figure 4 provides an overview of the flow of the Assertion Enforcement Mechanism. We assume that the Assertion Adopter has already registered to the Assertion Oracle and defined Assertions. That flow is not shown for brevity. The Assertion Adopter then registers with the Assertion Enforcement Mechanism and deposits funds for it's security budget, which are then used to pay out rewards to the Assertion Enforcer. If an Assertion Consumer submits a valid Proof of Realization, then the Assertion Enforcer is slashed.

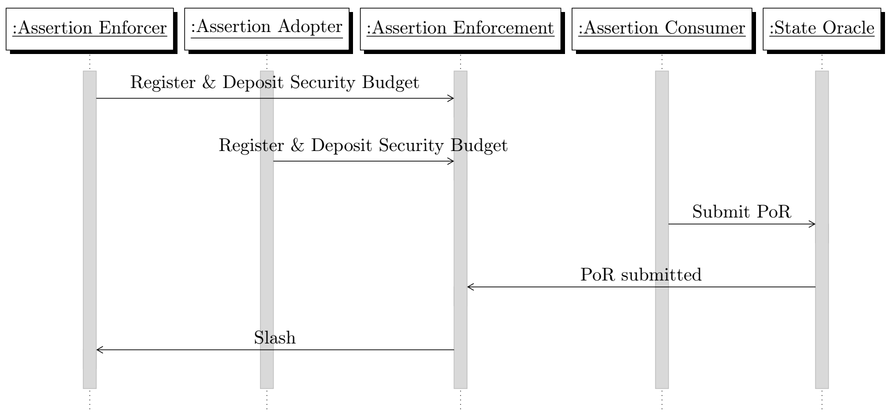

Figure 4: Assertion Enforcement Mechanism Flow

2.6 Pessimistic Path Proofs

A key insight into the design is that the system works optimistically for all intents and purposes. Assertion Adopters pay the Assertion Enforcers with the assumption that they are enforcing the assertions, without them having to prove it. Proofs are only posted on the pessimistic path, when the Assertion Enforcers allow an invalidating transaction to land on the chain.

This enables the architecture, discussed later, to optimize for performance, without the bottleneck of a computationally expensive proof generation process. There are no liveness requirements, as the proofs can be submitted at any point in the future, resulting to the penalization of the Assertion Enforcers.

This approach enables a strong separation of concerns between the crypto-economic related processes, such as fee distribution and slashing, and the assertion enforcement, which is optimized for performance.

2.7 Mechanism Desiderata and Security

When designing a mechanism, it's vital that one starts from the desiderata *(Latin: "things desired")*. As one can optimize for various properties such as trust-minimization, decentralization, complexity, and latency.

For instance, Optimistic mechanisms are trust-minimized, but usually introduce latency [1]. Honest-majority mechanisms are simpler and faster, but require trust to a quorum.

2.7.1 Desiderata

1. Latency: The mechanism should be able to add as minimum latency as possible to the underlying base layer.

2. Security: The mechanism should not allow a sophisticated attacker to bypass it and exploit a vulnerability of a dApp.

3. Simplicity: The mechanism should enable a simple architecture, and be as simple as possible. Simple systems are easier to understand, implement, most importantly audit.

4. Trust-minimization: The mechanism should remove as many trust assumptions as possible, without sacrificing the other desiderata.

2.7.2 Security Analysis

The security of the Credible Layer relies on a combination of economic incentives and social factors that discourage malicious behavior by Assertion Enforcers. Here are the key points:

Economic Incentives Assertion Enforcers are rewarded for honest behavior and face potential loss of their bond if they fail to prevent hacks. It's vital that the system offers both an ongoing reward for honest behaviour, as well as a strong deterrent for attacks. The game theory of a similar construct has been explored and proved to be functional in the case of Bitcoin [4].

However, the bond alone may not be enough to deter attacks involving large sums of money, as an Assertion Enforcer realistically cannot lock a bond comparable to the typical value of exploited funds, which often reach tens of millions of dollars.

From a purely crypto-economic standpoint, in the case of the most severe hacks, a hacker could offer a bribe far exceeding the bond of the Assertion Enforcer. On top of that, even if we assume that the Assertion Adopter will try to match that bribe, there is an upper limit to how much it can afford to pay, as the hacker is essentially paying with user funds, which the Assertion Adopter is not able to use.

Social Slashing The concept of social slashing introduces additional deterrents:

- Legal Ramifications: Facilitating a hack could result in criminal charges.
- Reputation Damage: Clear evidence of involvement in a hack would damage an Assertion Enforcer's reputation.
- Future Opportunity Costs: Participation in a hack would likely exclude an Assertion Enforcer from future involvement in the crypto ecosystem.

These social factors significantly increase the cost of dishonest behavior, making it unattractive even when the potential gains from a hack are large.

This is an important point to underline, as it shifts the game-theoretic model in a way that makes the mechanism functional. A hack is a very well defined event, which is considered as illegal in most if not all jurisdictions. A collusion between the Assertion Enforcer in a way that is cryptographically proven, as in the case of the Credible Layer, would result in significant real life consequences for the entities that are involved. The Assertion Enforcers are operated, as network providers, by organizations which will be directly liable for the realization of the hack.

Although this argument would hardly hold in the case of a blockchain network, where the "correct" state of the ledger is not defined clearly, it can very much hold in a system that is designed to prevent hacks. We leverage the nature of the underlying event to our advantage, as an input in our game-theoretic model.

Slashing as Insurance Thus, the slashing mechanism acts as a form of minimum insurance for Assertion Adopters. If an Assertion Enforcer fails to prevent a hack, the Assertion Adopter receives compensation through the slashed funds. The slashed bond is more valuable to the Assertion Adopter as a small coverage, than to the Assertion Enforcer who in the case of a collusion, they are compensated with the bribe.

2.7.3 Addressing the Desiderata

Latency The Credible Layer adds minimal latency to the underlying base layer as there is no consensus amongst different nodes for whether an assertion is invalidated or not after a transaction is executed. The 1 of 1 security model ensures that the system has the minimum possible latency, from a mechanism design perspective. Any further latency reduction is an engineering problem for the implementation, and is not a fundamental problem with the design.

Security The mechanism provides strong security guarantees by making it extremely difficult for attackers to bypass. Since Assertion Enforcers are responsible for adding transactions to the chain, any attempt to exploit a vulnerability must pass through their checks. The combination of economic incentives and social slashing creates a robust deterrent against collusion between Assertion Enforcers and potential attackers.

Simplicity The Credible Layer maintains a relatively simple architecture. The core mechanism revolves around Assertion Enforcers checking transactions against a set of assertions before including them in the chain. This straightforward approach makes the system easier to understand, implement, and audit compared to more complex security solutions.

Trust-minimization While the system does require trust in Assertion Enforcers, the social slashing mechanism significantly mitigates this trust assumption. The severe consequences of facilitating a hack, including legal, reputation, and future opportunity costs, create a strong incentive for honest behavior. This approach achieves a balance between trust-minimization and other desiderata, particularly latency and simplicity.

2.7.4 Assertion Enforcer Maleability

An interesting property of the Assertion Enforcer is that the mechanism doesn't make any assumptions about whether it's a single entity or a set of entities. While we assume that it's going to be a single entity, say a block builder, to minimize latency and thus stay competitive in the related base layer market (e.g block building), *that doesn't need to be the case.*

An Assertion Enforcer could very well be a quorum of nodes who come into consensus about whether a set of transactions are valid or not, through some Proof of Stake consensus mechanism. Such an approach obviously increases the latency of the Assertion Enforcer, but also the cost of corruption, even if in practice it's unlikely that a node would collude to facilitate a hack.

Such malleability is very useful in networks where there are multiple Assertion Enforcers, as it enables the Assertion Adopters to choose which Assertion Enforcers they want to trust and what tradeoffs they are willing to make in terms of latency, security, reputation and even security fees and bonded value.

2.7.5 Conclusion

The Credible Layer's design effectively addresses the key desiderata of latency, security, simplicity, and trust-minimization. By leveraging the unique nature of hacks as well-defined, illegal events, the mechanism creates a strong security model without sacrificing performance or introducing excessive complexity.

The social slashing concept, while challenging to formalize in traditional game-theoretic terms, provides a crucial layer of security that scales with the protected value. This approach allows the Credible Layer to offer robust protection even for high-value targets, without requiring excessive bonding or complex consensus mechanisms.

While the system does rely on trusting Assertion Enforcers to some degree, the combination of economic incentives and social slashing creates a security model that is resilient against collusion and large-scale attacks. The potential for severe real-world consequences serves as a powerful deterrent, effectively aligning the interests of Assertion Enforcers with the security of the network.

In summary, the Credible Layer achieves a balanced approach to blockchain security, offering strong protection against hacks while maintaining high performance and simplicity. This design demonstrates that by carefully considering the nature of the security threats and leveraging both on-chain and off-chain incentives, it's possible to create a highly effective security mechanism without sacrificing other crucial system properties.

2.8 Security Fees

Security fees constitute a fundamental component of the Credible Layer, serving dual purposes: mitigating spam attacks and generating revenue for Assertion Enforcers. The implementation of these fees is designed to be both efficient and fair, leveraging the computational nature of the assertions themselves.

2.8.1 Computational Resource Metering

The Credible Layer utilizes EVM bytecode for assertion expression, which inherently provides a basis for a computational resource pricing model. In this context, "gas" refers not to an on-chain fee, but rather to a metric quantifying the computational resources consumed by assertions. This metric enables precise calculation of fees that each Assertion Adopter must remit for the computational overhead incurred by Assertion Enforcers in enforcing assertions.

2.8.2 Dynamic Pricing Model

Through gas metering, the system implements a dynamic pricing model for both assertion execution and definition. This model scales fees according to the computational resources demanded from Assertion Enforcers by each Assertion Adopter. Given the unpredictable nature of transaction volume interacting with an Assertion Adopter, fees are collected post-facto. This necessitates an auxiliary mechanism to process relevant transactions, calculate fees, and facilitate their distribution. Moreover, given that fee calculation will most probably require proofs related to how many transactions were processed by the Assertion Enforcer (e.g included in a block) and interacted with the Assertion Adopter, the computational effort required to calculate the fees is non-trivial.

2.8.3 Fee Calculation and Distribution Approaches

Two primary design approaches are under consideration for fee calculation and distribution:

1. **Optimistic Collection**: This approach allows Assertion Enforcers to collect fees within predefined boundaries without submitting proofs. It incorporates a challenge mechanism whereby any party can submit proof of improper fee collection, resulting in slashing of the offending Assertion Enforcer.

2. **Oracle-based Collection**: This method employs a third-party oracle to calculate and distribute fees across the network to Assertion Enforcers. The oracle can be implemented using Trusted Execution Environments (TEEs) providing proof of valid execution, or as an economically secured, honest-majority oracle with staking mechanisms. Zero-Knowledge Proofs can also be used in place of TEEs, but at a higher computational cost and possibly infeasible for the system.

A sketch of the fee structure is provided in Appendix A.2, offering further insights into the economic model underpinning the Credible Layer.

2.9 Mechanism Extensions

The mechanism can be extended in a variety of crypto-economic ways. In the appendix we explore the following extensions, illustrating how the core mechanism can be used as a primitive for building new mechanisms on top of it:

- **Restaking Module**: The Credible Layer mechanism can be used as a restaking module, enabling validators to restake their staked ETH to Assertion Enforcers of their choosing. It reduces the capital intensity of the Assertion Enforcers, as they can source outside capital to bond to the mechanism. At the same time, it enables them to bond higher security budgets, as they don't have to provide the entire security budget themselves. Ultimately, it reduces the amount of trust that Assertion Adopters need to have in Assertion Enforcers, and delegates that trust to the validators who delegate their funds to the Assertion Enforcers. It is explored in Appendix A.3.
- **Bond Markets**: We can generalise the restaking module approach to a generic bond market. Assertion Enforcers, and other actors such as Assertion Submitters, can sell bonds which provide yield to the buyer that is linked to the actor's provable and dependable future fees. It is a way for the actors to get liquidity now, in exchange for a share of their future fees. It is explored in Appendix A.4.
- **Security Bond Markets as Insurance**: It is possible to utilise the bond markets as a medium to design an on-chain insurance program, where end-users pay fees to be covered while they use the Assertion Adopter's protocol, with other users taking the opposite side of the bet and receiving yield from the fees. The insurance is tied to specific assertions, thus there is no ambiguity regarding the policy coverage or the associated risk. It is explored in Appendix A.5.

3 Ajax Implementation

In this section we present the implementation of the first iteration of the Credible Layer, named **Ajax**. We discuss the components that are relevant to the mechanism design and explain their interactions. The first iteration is the only kernel of the fully-fleshed mechanism and includes the following components:

- Assertion Format, DA and execution (PhEVM)
- Assertion Enforcer specification and integration with OP-stack networks (OP-Talos)
- Credible Layer smart contracts
- Generation and verification of Proof of Realization and Proof of Possibility

Figure 5 illustrates the components and actors of the first iteration of the Credible Layer. Actors and components are organized into three layers: The User Layer, which is external to the underlying network, the dApp Layer which consists of the smart contracts that are deployed on the network and the base layer which consists of the actors serve as the infrastructure of the network (e.g Sequencers).

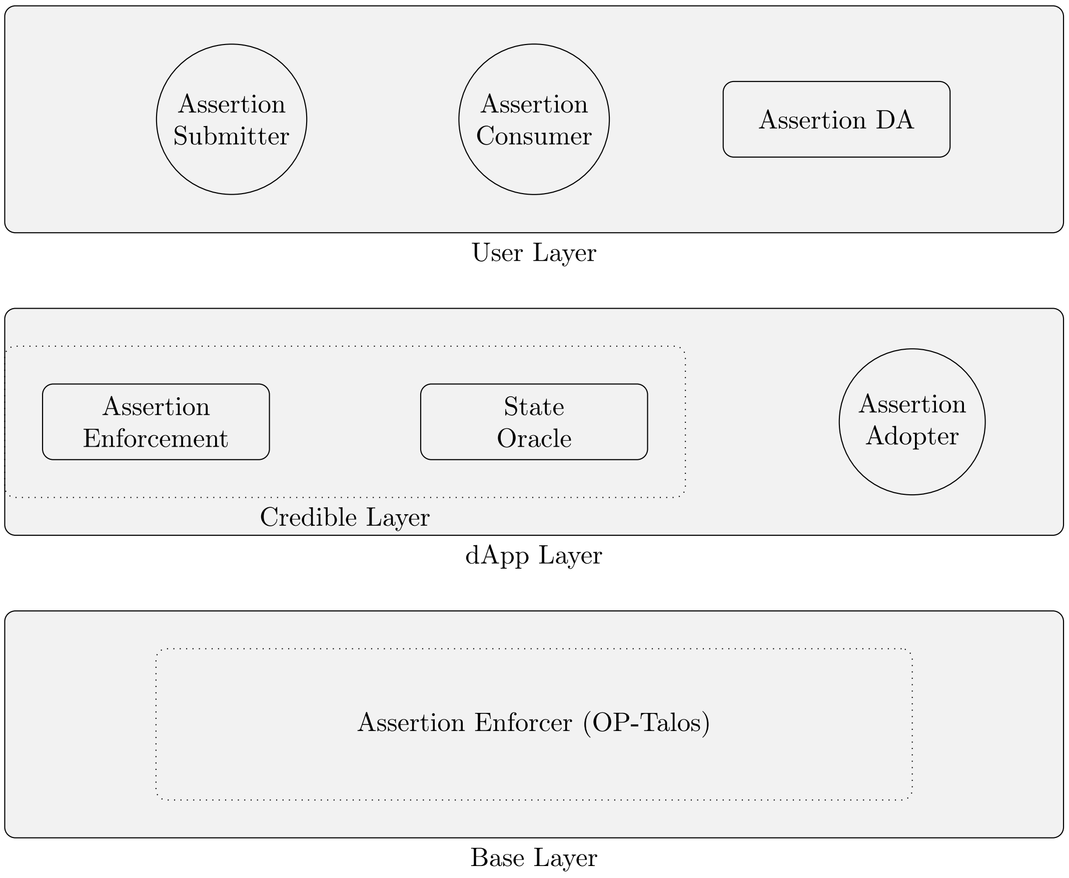

Figure 5: Ajax Implementation Overview

3.1 Assertion Format

Assertions are EVM [53] bytecode programs that are executed on a modified EVM interpreter, called the PhEVM. They are submitted to the Credible Layer in the form of raw bytecode that is then executed by Assertion Enforcers while validating transactions and by Assertions Consumers to construct proofs.

The Credible Layer omits bytecode validation, relying instead on the deterministic nature of PhEVM execution. For any invalid assertion bytecode, the PhEVM will consistently fail execution, preventing proof generation. This allows Assertion Enforcers to safely discard invalid assertions without additional verification overhead.

Thus, it's in the best interest of a Assertion Adopter to provide valid bytecode for their assertions. Maliciously crafting an invalid assertion would only result in the Assertion Adopter wasting security budget.

3.2 Assertion DA

Definition 3.1 (Assertion ID)**.** The Keccak256 Hash of the EVM bytecode of a *valid* Assertion.

Assertions require persistent storage accessible to both Assertion Consumers and Assertion Enforcers throughout their active lifecycle. Furthermore, the protocol must verify the integrity of the on-chain assertion ID by confirming it matches the Keccak256 hash of the assertion's EVM bytecode.

The current architecture employs a centralized off-chain server for assertion bytecode storage. This server, operated by Phylax Systems, issues signed assertion IDs to Assertion Adopters. The on-chain protocol validates the server's signature, thereby establishing the assertion ID's authenticity.

Post-initial implementation, a key priority is the decentralization of Assertion Data Availability (DA). We are exploring several potential designs:

1. Integration of DA within our own Credible Layer Network, which we may use to offer additional functionality, such as Assertion DA.

2. On-chain deployment of Assertions as smart contracts on the base layer, facilitating source code verification (e.g Etherscan). This approach presents a trade-off between ease of storage and verification versus iteration costs for Assertion Adopters, which may vary across networks.

3. Utilization of third-party DA networks such as Celestia [31], EigenDA [33], or Ethereum EIP-4844 blobs [41]. While this option introduces external dependencies, it leverages existing specialized infrastructure.

3.3 PhEVM

The PhEVM is a modified EVM interpreter with a number of precompiles that facilitate the execution of Assertions.

Currently, the PhEVM has access to:

1. **PreState**: The state before the transaction that triggered the assertion is applied

2. **Transaction**: The transaction that triggered the assertion

In future iterations, we will explore enabling state access to older historical states, but as this is a non-trivial engineering challenge given the performance requirements, we leave this for future work.

The native precompiles of the PhEVM enable the assertions to easily switch between the two states and also get secondary information, such as the transaction calldata that triggered the assertion or the logs that were emitted during the execution of that transaction.

Execution Flow

1. PhEVM forks at the state which triggered the assertion. Results to S_{forked}
2. PhEVM deploys the assertion contract on S_{forked}. Results to $S_{assertion}$
3. PhEVM loads the cached list of function selectors for the assertion
4. PhEVM calls all the assertion functions, in parallel, over $S_{assertion}$

If any assertion function reverts, the entire assertion is considered invalid and will be ignored. Figure 6 illustrates how the execution flow of the PhEVM, which happens right after the transaction is applied to the state. Assertions are loaded based on the contracts whose state changed as the result of the transaction.

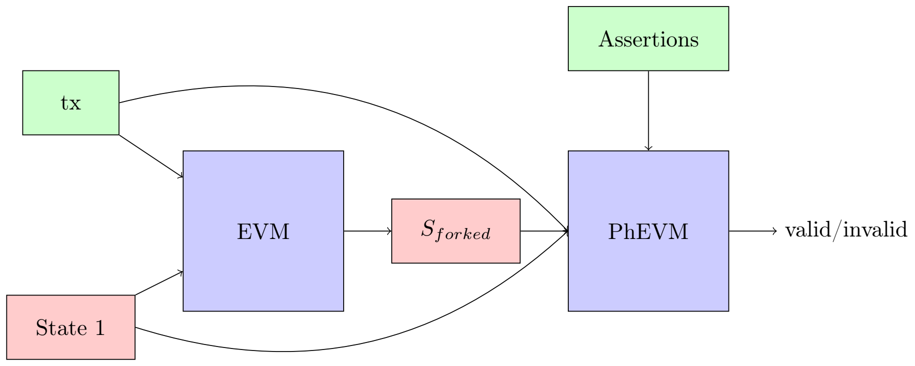

Figure 6: PhEVM Execution Flow

Finally, regarding the performance of PhEVM, there are many trivial optimizations we can do, such as Ahead-of-Time compilation of the assertion bytecode [38]. We don't expect the PhEVM to ever become a performance bottleneck.

3.4 Assertion Enforcer specification

The Assertion Enforcer is a critical component responsible for validating that transaction outputs do not invalidate assertions for a given Assertion Adopter.

3.4.1 Transaction Processing

Given an ordered group (e.g a block) of transactions $G = \{t_1, t_2, ..., t_n\}$, the Assertion Enforcer performs the following operations:

1. Initialize the simulation state S_0 to the state of the latest block in the canonical chain.

2. For each transaction $t_i \in G$, where $i \in \{1, ..., n\}$:
    (a) Apply t_i to the current state S_{i-1}, creating a new inter-block state S_i.
    (b) Execute all relevant assertions $A = \{a_1, a_2, ..., a_m\}$ against S_i.
    (c) If $\forall a_j \in A, a_j(S_i) = \text{true}$, persist S_i for the next transaction.

3.4.2 Execution Environment

- Regular transactions must be executed on an EVM identical to the underlying base layer.
- Assertions must be executed on the PhEVM.
- Inter-block state changes must persist within a single group of transactions but not between different groups.

3.4.3 Assertion Execution

Assertions are EVM bytecode programs that are executed on the PhEVM. They are executed post-transaction to validate desired outcomes. The Assertion Enforcer must execute all assertions associated with an Assertion Adopter if a transaction modifies the state of the Assertion Adopter.

3.4.4 Handling Invalidated Assertions

If $O(S_i) = \text{Invalid}$ for any S_i, the Assertion Enforcer:

1. **MUST** exclude the invalidating transaction t_i from the finalized group of transactions.
2. **MUST** restart simulation of the group $G' = G \setminus \{t_i\}$.
3. **SHOULD** initiate the generation of a Proof of Possibility (PoP) and submit it by sending a transaction at a later block.

3.4.5 Failure Scenarios

The PhEVM returns a binary outcome for each assertion: valid or invalid. Assertion Enforcer failures are handled according to implementation-specific fallback mechanisms.

3.4.6 Transaction Group Approval

A group of transactions G is considered approved if and only if:

$$\forall t \in G, \forall S_t \text{ resulting from } t : O(S_t) = \text{Valid} \tag{2}$$

The approved group maintains the original transaction order, and is subsequently passed to the appropriate network participant. For example, if the group of transactions is a block (so the Assertion Enforcer is a block builder), the block (group) is passed to the proposer for inclusion into the canonical chain.

3.5 OP-Talos

This subsection presents the implementation of the Assertion Enforcer within the OP-Stack framework, utilizing the experimental Builder API proposed by Flashbots [18][3]. At present, this implementation is constrained to rollups built on the OP-Stack that have implemented the emerging builder-proposer separation architecture, though no such networks currently exist in production.

Adopting the Builder API spec allows us to focus on building core parts of the Credible Layer for the initial iteration. The architecture is intended to be modular, so eventually we will have slightly modified versions of this middleware for each individual rollup.

3.5.1 Builder API Integration

To facilitate seamless interaction between the local block builder and the OP Stack sequencer, the following API endpoints are crucial:

- Fork choice update transmission
- `builder_getPayloadV1` support
- `builder_forwardTransactionV1` support (optional)

The `builder_forwardTransactionV1` endpoint is utilized for transaction forwarding from the sequencer mempool to the builder. This step may be omitted if the sequencer designates the builder as a trusted peer.

3.5.2 Block Building Process

The block building process in the Phylax Credible Layer architecture involves a complex interplay between various components.

Figure 7 illustrates this process.

[3]Our initial architecture resembled Rollup-Boost [21]; however, we opted for the Flashbots PBS specification to align with open standards. We continue to evaluate both approaches to determine the optimal solution for our network integrations.

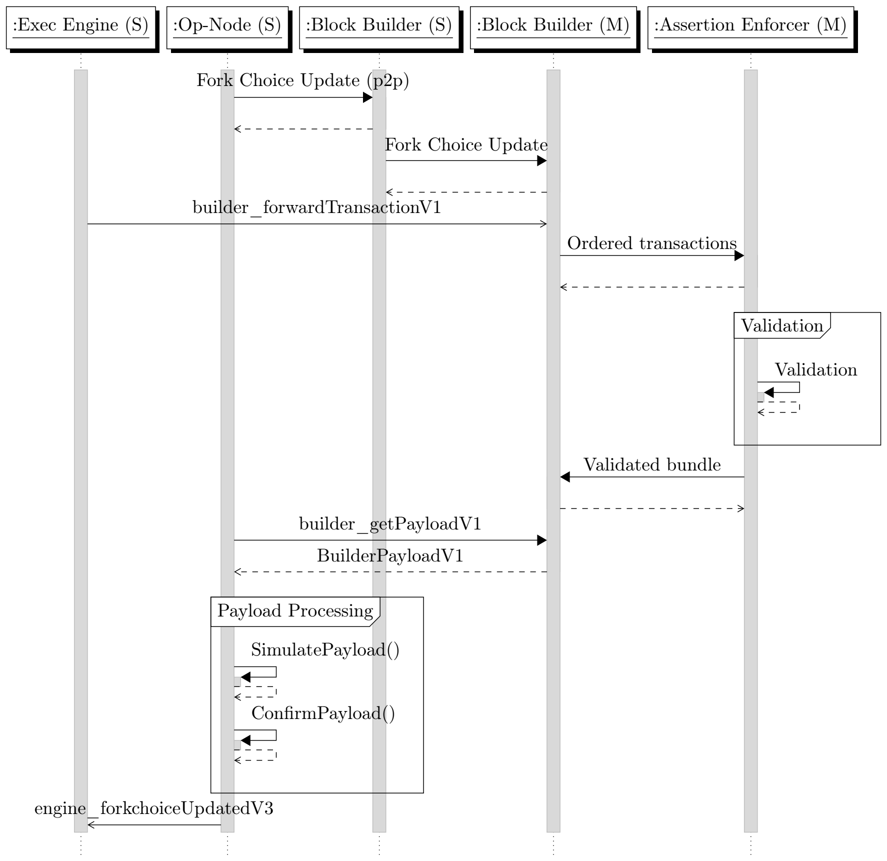

Figure 7: Sequencer-Builder-Assertion Enforcer Interaction Flow. Sequencer (S) is the Rollup Node, and Middleware (M) is the Assertion Enforcer Middleware.

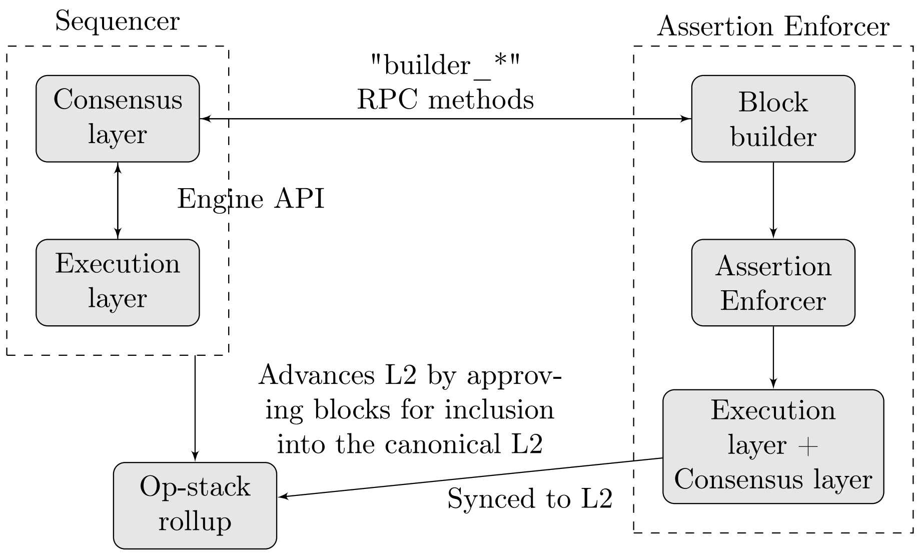

Figure 8: Credible Layer Architecture Flowchart

3.5.3 Failure Handling Mechanisms

In the event of Block Builder failures, OP-Talos implements two primary strategies to maintain network integrity:

1. Sequencer Fallback: In accordance with the PBS specification [18], if no response to `builder_getPayloadV1` is received within $200ms$, the Consensus Layer reverts to its internally built block. This approach ensures chain progression but may potentially include Invalidating transactions.

2. Empty Block Inclusion: Due to the nature of the Assertion Enforcer, allowing the Sequencer to build blocks without the Assertion Enforcer is dangerous, as it can include Invalidating transactions. To mitigate this, the system can be configured to produce empty blocks upon detecting Block Builder failures.

This is achieved by sending an empty `executionPayloadV1` to the Consensus Layer, thereby advancing the chain without including any potentially Invalidating transactions.

While the empty block approach effectively prevents the inclusion of Invalidating transactions, it may introduce disruptions to time-sensitive decentralized applications. Notably, it may cause oracle price desynchronization, impede liquidation processes in lending protocols, and potentially lead to unintended losses in DeFi ecosystems.

The implementation of a fallback builder, functioning as a proxy for the Credible Layer Block Builder, necessitates careful configuration to ensure seamless integration between the sequencer and the Credible Layer components.

3.5.4 Forced Inclusion of Invalidating Transactions

OP-Stack Rollups [37] (and others like Arbitrum [2]) have the concept of forced inclusion, where a transaction can be forced to be included in the rollup by submitting them to the host chain.

In this scenario, it seems that the transaction will be included in the rollup, without the Assertion Enforcer being able to do anything about it. In reality, there is a timelock delay of up to 12 hours before a transaction can be included in the rollup. This is ample time for the Assertion Enforcer, or any relevant stakeholder, to perform the following steps:

1. Identify the transaction as invalidating an assertion
2. Generate a Proof of Possibility (PoP)
3. Submit the PoP to the network and mark the mitigations related to the assertion that is invalidated as executable
4. Submit a transaction to the rollup which executes the mitigations

At this point, the mitigations have been executed and thus the inclusion of the invalidating transaction should not be able to inflict any damage.

Thus, if the Assertion Adopter has defined mitigations for the assertions, then there is no risk to the protocol. To be precise, even if they don't define mitigations, if they observe the forced transaction, they should be able to manually mitigate the issue before it's included.

3.6 Credible Layer Smart Contracts

The Credible Layer is deployed on every network it's integrated with. It's a set of smart contracts that perform the following functionality:

- State Oracle: Responsible for managing assertions, their state, the proofs and possible mitigations
- Assertion Enforcement: Responsible for managing Assertion Enforcement registration, rewards and slashing

The smart contracts are rather simple in their implementation, and don't warrant extensive discussion.

For the sake of completeness, the management of assertions can be delegated from the address that is used to register the Assertion Adopter to the mechanism. It enables a dApp to use the deployer private key and then delegate management of such a critical system to a multisig address or even Governance smart contract.

3.6.1 Assertion State

In order to tackle latency issues and address timing games, assertions have a certain delay to be activated in the protocol. This implementation detail doesn't change anything when it comes to the mechanism.

State Transitions

- Submitted: The assertion is submitted to the protocol.
- Active: The assertion is active and valid for PoR or PoP proofs.
- Removed: The assertion is removed from the protocol.

There is a timelock delay for both the submission and removal of an assertion. It enables the Assertion Enforcer to have time to fetch and cache the assertions, or mark them for removal. It protects against edge cases where an assertion would be invalidated but the Assertion Enforcer has not yet been able to check the protocol for the removal.

Finally, the timelock delay can be used by dApps in case the assertion management role is compromised. As an attacker could activate assertions that would render the protocol effectively unusable, the timelock delay enables the dApp to coordinate with users to remove assets before the activation delay expires. It's worth noting that in such a case, the Assertion Enforcer could voluntarily disable assertion enforcement to allow users to remove assets, with maybe the Assertion Adopter shouldering the slashing cost in case of Proof of Realization (PoR) invalidations.

3.6.2 Mitigations

A mitigation is a formal construct within the protocol, defined by the Assertion Adopter and associated with specific assertions. It serves as an executable response to validated vulnerabilities.

A mitigation has the following properties:

1. It is a valid encoded EVM call.
2. It targets a contract address that is:
   (a) A registered Assertion Adopter.
   (b) Managed by the same Assertion Adopter manager as the Assertion Adopter associated with the triggering assertion.

The execution of mitigations adheres to the following principles:

1. The success or failure of the mitigation call is not verified by the Credible Layer protocol.
2. Upon any invocation, regardless of outcome, the Credible Layer protocol marks the mitigation as executed.
3. Execution is permissible only after the associated assertion has been invalidated by a valid Proof of Possibility submission.

Remark. The design of the mitigation system prioritizes:

- Simplicity: Mitigations are native EVM calls.

- Robustness: The system is indifferent to call failures.
- Controlled activation: Proof of Possibility enables but does not enforce mitigation execution.
- Scope limitation: Mitigations can only target specific, related contracts within the protocol ecosystem.

3.7 Proofs in the Credible Layer

Proofs constitute a fundamental component of the Credible Layer, serving to verify the outcomes of off-chain computations, specifically the assertions executed in the PhEVM. These proofs are designed and implemented in accordance with the RISC-V ZKVM architecture. The process involves writing a program in Rust, compiling it to RISC-V machine code, and executing it on the ZKVM to generate a proof verifiable on-chain.

Both off-chain and on-chain components are implemented with modularity, enabling support for multiple ZKVM implementations, including SP1 by Succinct [45] and RISC0 [8], among others [50].

This section elucidates the two types of proofs utilized in the Credible Layer: **Proof of Possibility** and **Proof of Realization**.

Implementation details are omitted as they are straightforward and will be made available upon open-sourcing our codebase. We intend to provide both a Rust crate and a CLI tool to facilitate easy generation and submission of proofs to the protocol, ensuring accessibility for both end-users and security researchers.

3.7.1 Blockhash Oracle

EVM networks natively provide access to the block hashes of the 256 most recent blocks [53]. These block hashes serve as the root of trust for the blockchain state at a given block. Proofs verify the storage proofs of all accessed storage slots during PhEVM execution and subsequently verify the state root against the block hash.

Given the limited temporal scope of 256 blocks, an extension mechanism is necessary to increase the availability of block hashes over a more extended period. While solutions like Axiom [5] and Herodotus [25] employ ZK proofs, we have opted for a simpler approach. We implement a smart contract utilizing a Merkle Mountain Range (MMR) structure [35], which incorporates all available block hashes (from the last 256 blocks) into the tree. Regular invocation of this smart contract extends the number of available block hashes.

Our analysis indicates that for most rollups, such as Base, the cost of invoking the MMR smart contract is negligible. For more resource-constrained networks like Ethereum, while the cost is more significant, off-the-shelf solutions like Axiom can be utilized.

It is noteworthy that this functionality could potentially be integrated into the ZK primitives we will be constructing, which would eliminate liveness concerns associated with the blockhash oracle. While this integration remains a subject for future research, the current implementation provides a simple and effective solution.

3.7.2 Proof of Possibility (PoP)

Let $\mathcal{A}$ be the set of addresses, $\mathcal{H}$ the set of 256-bit hashes, $\mathbb{N}$ the set of natural numbers, $\mathcal{T}$ the set of transactions, and $\mathcal{B}$ the set of bytecode.

$$\begin{aligned} ProofGen &: \mathcal{A} \times \mathcal{H} \times \mathbb{N} \times \mathcal{T} \times \mathcal{B} \times \mathcal{A} \to \mathcal{P} \\ ProofVerify &: \mathcal{A} \times \mathcal{H} \times \mathbb{N} \times \mathcal{H} \times \mathcal{H} \times \mathcal{A} \times \mathcal{P} \to \{0,1\} \end{aligned} \tag{3}$$

Where $\mathcal{P}$ is the set of valid ZK proofs.

PoP Generation Inputs

- $d \in \mathcal{A}$: Assertion Adopter's address
- $h \in \mathcal{H}$: Blockhash of the block
- $n \in \mathbb{N}$: Block number
- $\vec{t} \in \mathcal{T}^*$: Vector of invalidating transactions
- $c \in \mathcal{B}$: Bytecode of the assertion
- $s \in \mathcal{A}$: Address of the PoP submitter

It's worth noting that we generate the proof over a vector of transactions, as there might be multiple transactions that could cause the invalidation of the assertion. For example, there is a number of vulnerabilities where the attacker needs to "prepare" the hack, ahead of the actual exploit.

PoP Generation Flow

1. Get blockchain state S_n at block n
2. Get block B_n
3. Run PhEVM with inputs: state S_n, transaction $\vec{t}$, and assertion code c
4. Note the accessed state and fetch storage proofs for the accessed storage slots
5. In ZKVM:
    (a) Verify the validity of B_n

   (b) Run PhEVM with inputs: S_n, $\vec{t}$, and c
   (c) Assert execution returns true
   (d) Verify storage proofs
   (e) Compute assertion id $a_{id} = H(c)$ and transactions hashes $\vec{t_h} = H(\vec{t})$
   (f) Verify h corresponds to block n
   (g) Constrain: d, h, n, $\vec{t_h}$, a_{id}, s

6. Generate and return proof $p \in \mathcal{P}$

PoP Verification Inputs

- The values that were constrained in the Proof Generation: d, h, n, $\vec{t_h}$, a_{id}, s
- $p \in \mathcal{P}$: The generated ZK proof

PoP Verification Flow

1. Verify h is a valid historical blockhash
2. Verify a_{id} is valid for Assertion Adopter d and enrolled in the protocol
3. Verify that s is the address of the sender of the transaction that submitted the proof. This offers front-running protection.
4. Verify ZK proof p given public inputs
5. Return result $\in \{0, 1\}$

Where $H(\cdot)$ is the Keccak256 hash function.

3.7.3 Proof of Realisation (PoR)

Let $\mathcal{A}$ be the set of addresses, $\mathcal{H}$ the set of 256-bit hashes, $\mathbb{N}$ the set of natural numbers, $\mathcal{T}$ the set of transactions, and $\mathcal{B}$ the set of bytecodes.

$$\begin{aligned} ProofGen &: \mathcal{A} \times \mathcal{H} \times \mathbb{N} \times \mathcal{T} \times \mathcal{B} \times \mathcal{A} \to \mathcal{P} \\ ProofVerify &: \mathcal{A} \times \mathcal{H} \times \mathbb{N} \times \mathcal{H} \times \mathcal{H} \times \mathcal{A} \times \mathcal{P} \to \{0, 1\} \end{aligned} \tag{4}$$

Where $\mathcal{P}$ is the set of valid ZK proofs.

PoR Generation Inputs

- $d \in \mathcal{A}$: Assertion Adopter's address
- $h \in \mathcal{H}$: Blockhash of the invalidation block
- $n \in \mathbb{N}$: Block number of the invalidation
- $t \in \mathcal{T}$: Transaction causing the invalidation
- $c \in \mathcal{B}$: Bytecode of the invalidated assertion
- $s \in \mathcal{A}$: Address of the PoR submitter

PoR Generation Flow

1. Get blockchain state S_n at block n
2. Get block B_n
3. Apply transaction t to state S_n
4. Execute PhEVM with inputs: state S_n, transaction t, and assertion code c
5. Note the resulting state S_{n+1} and fetch storage proofs for the accessed storage slots
6. In ZKVM:
   (a) Verify the validity of B_n
   (b) Run PhEVM with inputs: S_n, t, and c
   (c) Assert execution returns true
   (d) Verify storage proofs
   (e) Compute assertion id $a_{id} = H(c)$ and transaction hash $t_h = H(t)$
   (f) Verify h corresponds to block n
   (g) Constrain: d, h, n, t_h, a_{id}, s
7. Generate and return proof $p \in \mathcal{P}$

PoR Verification Inputs

- The values that were constrained in the Proof Generation: d, h, n, t_h, a_{id}, s
- $p \in \mathcal{P}$: The generated ZK proof

PoR Verification Flow

1. Verify h is a valid historical blockhash
2. Verify a_{id} is valid for Assertion Adopter d and enrolled in the protocol
3. Verify that s is the address of the sender of the transaction that submitted the proof. This offers front-running protection.
4. Verify ZK proof p given public inputs
5. Return result $\in \{0, 1\}$

Where $H(\cdot)$ is the Keccak256 hash function.

3.8 Credible Layer in Multi-Builder Networks

Our previous discussions of the Credible Layer assumed a single Assertion Enforcer or block builder for the entire network. However, the evolving landscape of L2s, L3s, and Ethereum mainnet suggests a future where multiple block builders coexist [52]. Either in the form of multiple block proposers for a centralised sequencer or in the form of multiple sequencers for a single rollup (Shared Sequencing). This subsection explores how the Credible Layer architecture can adapt to such a multi-builder environment.

For clarity, we use the term "block builder" to encompass both traditional block builders on Ethereum mainnet and sequencers in L2 networks, as they share the fundamental responsibility of transaction inclusion.

Definition 3.2 (Credible Builder)**.** A Credible Builder is a block builder that is enrolled in the Credible Layer and acts as an Assertion Enforcer.

Definition 3.3 (Credible Block)**.** A Credible Block is a block that is built by a Credible Builder.

In a multi-builder ecosystem, the Credible Layer operates by:

1. Establishing a registry of credible block builders
2. Enabling dApps to verify whether a block is credible or not, reverting all transactions from non-credible builders
3. Creating incentives for builders to participate in the Credible Layer ecosystem

A key insight is that universal participation of block builders is not necessary. If a dApp rejects transactions from non-integrated block builders, the security model remains intact, even if some blocks contain potentially invalidating transactions.

Incidentally, this also solves the issue we mentioned before around the existence of private mempools and how they restricted the efficacy of current hack prevention solutions. No matter how many private mempools exist or how often validators collude with hackers to include malicious transactions, as long as they are not credible block builders, they will not be able to include non-reverting transactions that interact with some Assertion Adopter.

3.8.1 Block Builder Registry

The foundation of this system is a registry of block builders that have registered as Assertion Enforcers and posted the required bond:

Smart contracts require a mechanism to verify whether a block originates from a Credible Layer Assertion Enforcer. We propose two potential approaches:

1. **Top-of-Block Signaling Transaction Method:** This approach involves including a transaction in Credible Layer-protected blocks which "signals" that the block is credible. Depending on the consensus mechanism of the base layer, it can be potentially vulnerable to unbundling attacks [12], where a block is unbundled and the signaling transaction is added to a non-credible block.

2. **Block.Coinbase Method:** This method leverages the `block.coinbase` variable available in the EVM.

Note that the `block.coinbase` method assumes this variable accurately represents the block builder, which may not always hold in Ethereum's current implementation. Currently, this is not enforced by the protocol [49], but it is a well respected best practice.

In listing 1 we show the implementation of the Credible Layer Block Builders Registry, as well as the two methods to verify if a block is credible. More specifically:

1. `updateBuilder(address builder, bool isApproved)`: Enables governance to update the registry.

2. `isCredibleBlockBySignal()`: Verify if a block is credible by checking if the block number is the same as the latest credible block. It is used in the top-of-block transaction method, so the block builder must have called `signalCredibleBlock()` at the start of the block.

3. `isCredibleBlockByCoinbase()`: This function is used to verify if a block is credible by checking if the block builder is in the registry. It is used in the `block.coinbase` method, so the block builder must have been approved in the registry.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.27;

contract CredibleLayerRegistry {
    mapping(address => bool) public approvedBuilders;
    address public governance;
    uint256 public latestCredibleBlock;

    modifier onlyGovernance {
        require(msg.sender == governance, "Not authorized");
        _;
    }

    function updateBuilder(address builder, bool isApproved)
        external
        onlyGovernance
    {
        approvedBuilders[builder] = isApproved;
    }

    function isCredibleBlockByCoinbase() external view returns(bool) {
        return approvedBuilders[block.coinbase];
    }

    function isCredibleBlockBySignal() external view returns(bool) {
        return latestCredibleBlock == block.number;
    }

    function signalCredibleBlock() external {
        require(approvedBuilders[msg.sender], "Not authorized");
        latestCredibleBlock = block.number;
    }
}
```

Listing 1: Credible Layer Block Builders Registry

3.8.2 Incentive Structure

We can model the revenue differential between Credible Layer-integrated block builders and non-integrated ones as follows:

$$\Delta R_{builder} = F_{security} + V_{ExMEV} - C_{AE} \tag{5}$$

Where:

- $\Delta R_{builder}$ is the revenue difference between a standard block builder and a Credible Layer block builder
- $F_{security}$ is the security fee paid by Assertion Adopters
- V_{ExMEV} is the value extracted through exclusive MEV opportunities of the Assertion Adopter
- C_{AE} is the cost shouldered by the Assertion Enforcer associated with running assertions (computation, storage, latency hit in identifying the most profitable bundle, cost in case top-of-block transaction method is used, etc.)

We posit that the revenue differential $\Delta R_{builder}$ is positive, which means that it is at least marginally more profitable to be a Credible Layer block builder than a standard block builder. Such a statement requires proof, which we plan to provide in future work as we further explore how the Credible Layer exists in a multi-builder ecosystem.

3.8.3 Liveness Considerations

The proposed design introduces a potential liveness issue for Assertion Adopters in a decentralized block-building environment. By restricting the set of block builders that can interact with a Assertion Adopter, we create a scenario where the Assertion Adopter's liveness depends not on the overall network liveness, but on the liveness of Credible Layer-integrated block builders.

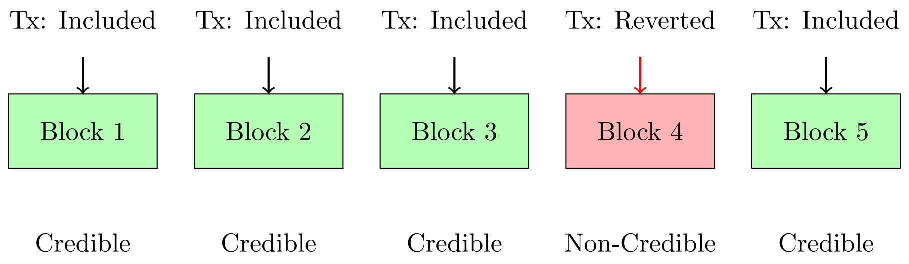

Figure 9: Transaction Inclusion in Credible vs Non-Credible Blocks

This risk can be mitigated by ensuring a sufficient number of integrated block builders. In practice, even a small number (e.g., three) of major block builders could provide adequate coverage. For example, on Ethereum Mainnet, over the last 30 days, as of October 5th 2024, the top three block builders produced over 97% of all the blocks [16]. Of course, we understand that there are multiple UX issues here, such as a user sending a transaction in the public mempool, which is picked up by a non-Credible Builder and included in the block, even if it reverts, forcing the user to resend.

A more challenging issue arises for dApps that require frequent interactions, such as lending pools and on-chain oracles. These applications may become out-of-sync if their update frequency is reduced, potentially disrupting arbitrage-based rebalancing mechanisms. We are actively researching solutions to this problem, but an initial way to tackle this could be by limiting the Credible Layer integration to critical code paths (e.g., smart contract upgrades) or redesigning the smart contract so that code paths that require frequent interactions are not affected by the liveness issue.

3.8.4 Credible Bundles

A variation of the above approach is to mark bundles of transactions, instead of a whole block. This would be useful for alternatives architectures, such as application-specific sequencing [55], where parties have control over a sequence of transactions, but not the entire block.

That actor would need to add two signaling transactions to the bundle: one at the beginning and one at the end. The first one would indicate that the bundle is the beginning of a new sequence of transactions, and the second one would indicate that the bundle is the end of a sequence of transactions. A naive way to do this would be to have a single transaction that sets a flag, in a smart contract like the registry mentioned above, to true at the beginning of the bundle and then another transaction that sets the flag to false at the end of the bundle. Figure 10 illustrates this concept.

Conversely, the Assertion Adopter's smart contract needs to check that the flag has been set at the entrypoint of the function call. That means that a transaction that interacts with the Assertion Adopter will revert, unless it is part of a this bundle of transactions.

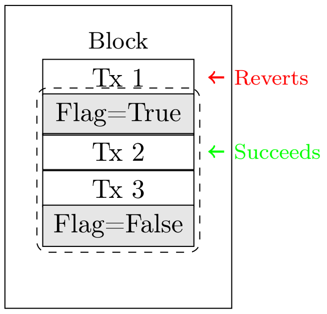

Figure 10: A block with a Credible Bundle: Transactions marked as "safe" by the Assertion Enforcer. The Assertion Adopter reverts transactions outside this bundle.

3.8.5 Conclusion

In this section we have discussed how the Credible Layer can be adapted to a multi-block builder environment, such as is the case for L1 networks, rollups with decentalised sequencing or rollups with shared sequencing.

The core insight is that the Assertion Enforcer needs to be a party that can vouch for a group of transactions. In reality, the Assertion Enforcer doesn't need to have control over the entire block building process, but over a group of transactions. To be more precise, the Assertion Enforcer doesn't even need inclusion control, but rather *exclusion control*. That is, the Assertion Enforcer can be a party which have the ability to veto transactions from being included in the canonical chain, such as the infrastructure providers for application-specific sequencing [32] or the Relays in Flashbots MEV-Boost [20].

4 A Sketch on Token Incentives

The mechanism, as discussed in Section 2, requires the transfer of value between parties in order to align their incentives and provide security to the Assertion Adopters.

In this section, we will discuss the various token types that can be used in the incentive design of the mechanism, as also how the protocol can capture value from the system and lead into a viable economic model to support the development of the protocol.

A token is required for the following transfers of value:

1. Assertion Adopters pay Assertion Enforcers for the execution of assertions (Security Fee)
2. Assertion Adopters pay Assertion Consumers for submitting Proofs of Possibility (Bounty)
3. Assertion Enforcers are slashed on successful submissions of Proofs of Realization (Assertion Enforcer slashing)

To support the above, the mechanism requires participants to hold some value in the system, so that the above transfers can happy when appropriate:

1. Assertion Adopter Security Budget: Assertion Adopters need to deposit funds so that they can be used to pay for the security fees. From these funds, a subset is **locked** to pay for the bounties that have been defined by the Assertion Adopter.
2. Assertion Enforcer security Budget: Assertion Enforcers need to deposit and **lock** funds, so they can be used for Assertion Enforcer slashing

Moreover, a token can be used for the following auxiliary systems that are required to support the mechanism:

1. Protocol Governance: Change parameters of the protocol or upgrade the code.

2. Assertion Data Availability: Assertion bytecode is required to be available to both the Assertion Enforcers for execution, as also the users to generate the proofs.

3. Security Fee calculation: Security Fees need to be calculated for every Assertion Adopter, on a continuous basis.

4. Other systems: As the protocol is developed, new infrastructure need mechanism incentives, such as what needed for *Cross-Chain Assertions*.

4.1 Token Types

Different types of tokens can be used either as a whole to achieve the above requirements, or in some combination thereof:

1. Phylax Native Token: An ERC20 token that is minted/burned by the Credible Layer Protocol according to the tokenomics of the protocol.

2. Network Native Token: The native gas token of the Network.

3. External ERC20 Tokens: An ERC20 token that is external to both the Credible Layer and the Network, such as LRTs or Stablecoins.

For the requirements of the sketch, we will make a few working assumptions:

1. Phylax Native Token: Potentially volatile and challenges the credible neutrality of the Network, as the Credible Layer is fundamentally integrated into the base layer. Enables flexibility with the mechanism design by controlling the levers of emissions and burn.

2. Network Native Token: Less volatile than the Phylax Native Token. Credibly Neutral for the base layer, as it is the token used for dApps and Users to interact with the Network. Inflexible as far as mechanism incentives are concerned.

3. External ERC20 Tokens: Potentially volatile (dApp Governance token), less volatile (LRTs) or not volatile at all (Stablecoins). Can be credibly neutral (LRTs), or not (Stablecoins). Inflexible as far as mechanism incentives are concerned.

4.2 A First Approach

To reformulate, we consider the following parameters for our problem:

1. Token Options: Phylax Native Token, Network Native Token, ERC20

2. Token Uses: Security Fees, Assertion Enforcer Slashing, Bounties, Auxiliary Services

As every network operates as a sovereign entity led by an organisation with active input from the community, we believe that modularity is key to the design, so that it can be flexible enough according to the needs of every network. Given that sovereignty, it only makes sense for them to be in control of this fundamental mechanism as well. A flexible design enables networks to configure the mechanism parameters according to their liking, such as the slashing amount, promoting competition amongst networks.

To that effect, we establish the following principles:

Token Use	**Token Used**	**Rationale**
Security Fees	ANY	The Network should have the freedom to choose any of the above token options. Incentives will be given by the Credible Layer Protocol to choose the Phylax Native token.
Assertion Enforcer Slashing	ANY	The Network should have the freedom to choose any of the above token options. Incentives will be given by the Credible Layer Protocol to choose the Phylax Native token.
Bounties	ERC20	ERC20 tokens offer flexibility and are widely accepted, making them ideal for bounties. Assertion Adopters can choose to pay out in their own governance token or use something less volatile (e.g. USDT) as a competitive advantage.
Auxiliary Services	Phylax Native Token	Utilizing the native token for auxiliary services, enables the Credilble Layer to be a profitable and economically viable protocol that is actively supported and developed.

Table 1: Token Uses and Rationale

The above selection enables a set of "social contracts", aligned with current Modus Operandi:

1. Networks are in control of how they are slashed and how their resources should be priced. If dApps are not happy with the configuration, they are incentivized to choose a new network to be deployed.

2. dApps are in control of how their assertions should be priced for bounties. If Security Researchers are not happy with the bounties, they have the incentive to analyze and submit Proofs of Possibility for other Assertion Adopters.

Regarding the locked funds, it is possible for the protocol to support a version of "external yield", so that the capital efficiency of the system is improved. Similar to how Blast [7] offers incentives, the protocol could enable the support of various DeFi protocols, such as Lido. Although, this opens up the system to second-order effects, such as the external DeFi protocol getting hacked (if not deployed of course in a Credible Layer-enabled network), it could be an option that both Assertion Adopters and Assertion Enforcers would find enticing.

4.3 Token Modularisation

The way that token modularisation would work is quite simple. The protocol would enable every network to control the parameters of the Slashing and Security Fee formulas, via their on-chain definition. Then, the security fee calculation "service" would be compelled to use these parameters when calculating the amounts. While the form of the formula is dictated by the protocol, the exact parameters are not, enabling different pricing according to the value of the token used.

As an example, consider the fee formula illustrated in Appendix A.2. The various parameters could very well be put on-chain on the Credible Layer smart contracts and defined by the network at deployment time. These parameters would then be respected while calculating the Security Fee for that particular network.

4.4 Tax

With the implementation of some form of tax in the Credible Layer protocol, the protocol can capture a small percentage of the value that is provided from the protocol itself. This tax could be then directed to the operators of the auxiliary services, enabling the Phylax Native Token to capture value and offer a superb UX to the users of the protocol. The UX improvement comes from the fact that if a tax is implemented, then Assertion Adopters and Assertion Enforcers don't need to hold the native token to leverage the auxiliary services and direct value to the operators of the auxiliary services. The value is indirectly captured and directed from the tax, which is distributed in some fashion to the operators of the auxiliary services.

4.5 Auxiliary services

The auxiliary services are needed to provide various guarantees to the users of the Credible Layer, such as the assertion data availability, but are not required for the core functionality of the protocol.

This enables every network to choose which services they want to leverage and how integrated they want the Credible Layer to be in their Network.

At their core, the auxiliary services will probably function as a network, such as an honest majority oracle where the quorum is decided based on the stake of the participants. In return, the participants will be rewarded with yield which originates from the users of the services, such as the Assertion Adopters and Assertion Enforcers.

The emissions can be broadly classified into two categories:

1. Internal Emissions: Emissions that are directed to the parties that participate in the Credible Layer Network and offer the auxiliary services.

2. External Emissions: Emissions that are directed to the parties that leverage the Credible Layer for security, such as Assertion Adopters, Assertion

Enforcers or even the users of the Assertion Adopters protocols.

4.5.1 Internal Emissions

Internal Emissions are directed to the parties that participate in the Credible Layer Network and offer the auxiliary services. They would need to stake some tokens, and the emissions correlate to the staked funds.

4.5.2 External Emissions

External Emissions are directed to the parties that leverage the Credible Layer for security, such as Assertion Adopters, Assertion Enforcers or even network users who interact with the Assertion Adopters. A few outcomes that these incentives would want to drive are:

1. Assertion Enforcer security budget: Incentivize networks to increase their security budget.
2. Assertion Adopter usage: Incentivize users to interact with protocols that are protected by the Credible Layer.
3. Bounty usage: Incentivize Assertion Adopters to assign bounties to their assertions, which boosts the security of the ecosystem as it incentives actors to find and submit vulnerabilities.
4. Phylax Native token: Incentivize Assertion Adopters and Assertion Enforcers that choose to use the Credible Layer token as the underlying token in the mechanism, as described above.

4.6 Conclusion

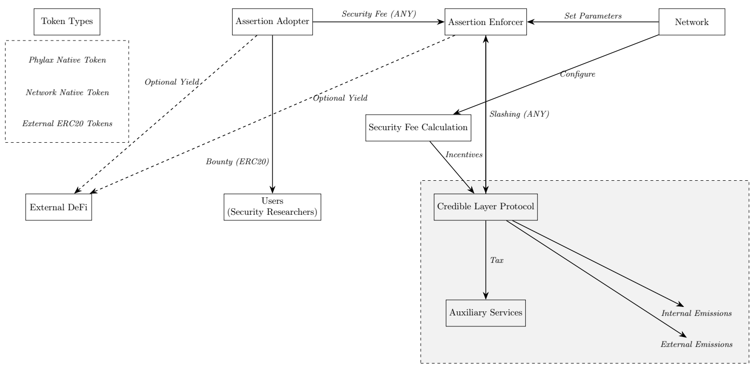

Figure 11: Comprehensive Tokenomics Flow and Components

The proposed design facilitates a flexible and modular approach to the tokenomics of the Credible Layer protocol, allowing individual networks to tailor the mechanism to their specific requirements. This customizability is crucial for promoting the adoption of the Credible Layer as a credibly neutral mechanism and essential infrastructure for the base layer.

The Credible Layer Architecture's organization into a modular core and auxiliary services enables customization of the core mechanism without compromising the protocol's overall viability. The protocol captures value through a combination of a tax on the core mechanism and the utilization of auxiliary services by networks, **without mandating** the use of these auxiliary services.

Consequently, a portion of the captured value is redirected to the auxiliary services and ultimately to holders of the Phylax Native Token. This structure not only ensures the protocol's financial sustainability but also allows for the implementation of incentive mechanisms aligned with the protocol's objectives, such as distributing Phylax Native Tokens to Assertion Enforcers maintaining high security budgets.

5 The Credible Layer Relationship with the Base Layer

In this section we will explore the design space of the Credible Layer, how it can be analyzed in the context of a network's Fork Choice Rule, and how we can approach it as enshrining off-chain computations in the base layer.

5.1 Definitions

First, we need to introduce the concept of a *Fork-Choice Rule*, a *Soft Fork*, and a *Hard Fork*.

Definition 5.1 (Fork-Choice Rule - FCR)**.** The Fork Choice Rule is the mechanism that a blockchain node uses to decide the canonical history of the blockchain, according to its local view of the network [26]. The FCR is executed continuously as time passes and the node learns new information about the network, against new and old blocks. If enough nodes on the network agree on a different sequence of blocks(as a result of the continuous execution of the FCR), the chain can reorg, where some blocks are removed from the canonical chain.

Definition 5.2 (Soft Fork)**.** A soft fork is a backward-compatible change to the blockchain protocol that allows new rules to be introduced without requiring all users to upgrade their software [44].

Definition 5.3 (Hard Fork)**.** A hard fork is a non-backward-compatible change to the blockchain protocol that requires all users to upgrade their software in order to continue participating in the network [44].

5.2 Integrating the Credible Layer with a Network

The Credible Layer can be integrated with a blockchain network in two primary ways: as a filter (per-node basis) or as a network-wide modification (hard fork).

Moreover, we can integrate the Credible Layer in a way that it modifies either the STF or the FCR of the network.

Essentially, not accepting a transaction in the first place means that effectively the STF is modified, resulting to the end-state being now invalid. On the other hand, if a malicious transaction is included in the canonical chain, but then later the transaction is removed from the chain due to a reorg, then we can say that we have modified the FCR.

Definition 5.4 (Credible Layer as a Filter)**.** A per-node integration where individual nodes run an Assertion Enforcer, modifying their local STF for block creation without altering the underlying protocol.

Definition 5.5 (Credible Layer as a Hard Fork)**.** A network-wide integration that modifies the STF of all nodes, enabling contract-based soft-forking through assertions submitted by Assertion Adopters.

The filter approach creates a dual STF for each node:

1. A modified STF for validating self-generated blocks, filtering out invalid transactions.
2. The original STF for processing blocks from the chain, maintaining network consistency.

In contrast, the hard-fork approach:

1. Modifies the STF network-wide to incorporate assertion-based validity rules.
2. Functions as a contract-based soft-fork once implemented, meaning that every contract can affect a soft-fork to the network by adding assertions, which modifies the underlying STF of the network as a whole.

We say that the hard-fork approach implements contract-based soft-forking, because the definition of new assertions effectively changes the network-wide STF, without the need for any additional changes to the protocol.

We don't consider the filter approach to enable contract-based soft-forking because it's not a network-wide change, but only local to the node who acts as an Assertion Enforcer and implements the filter.

5.3 Enshrining Hack Prevention in Rollups

In this section we will explore how the Credible Layer can potentially be used as an enshrined change of a rollup, either by modifying the FCR or the STF.

A rollup is a blockchain which inherits security [10] from the host-chain. It was first introduced in 2019 as "Merged Consensus" [1]. It is essentially a subset of another consensus, with a superset of state, as the rollup node needs to keep both the rollup state and the host chain state, and transition both [40].

The modification of the STF is forward-looking, as it doesn't allow for an improper state transition to be executed. The modification of the FCR is instead backward-looking, as it points a particular transaction as invalid, and removes it from the canonical chain after the fact.

The first approach prevents hacks alltogether, while the later removes the hack from the chain. The end-state is the same, the state transition that is considered invalid does not become, in the end, part of the canonical chain.

5.3.1 Enshrining the Modified FCR

rollups, as blockchains, have an FCR of their own. The Init4 research collective recently published an essay [26] about this subject, but effectively there are two groups of FCR designs:

1. Host-following: *All* rollup reorgs are triggered by a host chain reorg.
2. Host-watching: *Some* rollup reorgs are triggered by a host chain reorg.

Most rollups that exist on Ethereum, like Optimism, Arbitrum or the proposed designs for based rollups are host-following. What's relevant to the construct we have been discussing are *Host-watching* rollups, of which there are currently none.

As Init4 states:

> Like any fork-choice rule, the rollup FCR must rely only on information that the node can observe. In addition, rollups must be fully derivable from the host history. This imposes an additional constraint: rollup FCRs may only read information computable from the host history or state. This ensures that new nodes may still sync the rollup. This seems like a serious restriction, however it allows significantly more flexibility than it appears. Anything that can be submitted to host DA or produced by a smart contract can be used by the rollup FCR. This allows oracles, zk or fault proofs of any computation, DEX prices, and NFT ownership to be included in the rollup fork choice rule as long as they pass through the host history and/or state.

Init4 suggests a design [26] where the rollup FCR enables the removal of transactions from the canonical chain, if some party signals them as a hack, for example via a governance approach.

This concept is very easily supported by the State Oracle that we posed in Section 2. Assertion Adopters, with our without the help of Assertion Submitters, submit assertions to the State Oracle. Then, when a Proof of Realization is submitted, the State Oracle can be whitelisted to be able to call a function on the host smart contracts which signals that a particular transaction should be removed from the canonical chain. This can be easily implemented using the mitigations construct we described in the same section.

After the proof of submitted and the mitigation calls the rollup smart contracts, the rollup nodes will pick up the event and the FCR will reorganize the chain, removing the Invalidating transaction in the process.

Tradeoffs While it is feasible to implement various state modifications to maintain the validity of remaining transactions (e.g., resetting the Account Nonce of subsequent transactions), such removals may have unforeseen consequences on the dependent state. For instance, if a hacker converts their funds to a different asset, this could create an imbalance on a decentralized exchange, which might then be leveraged by an arbitrageur for profit. Consequently, the arbitrageur's trade would also need to be reverted, or else it could result in financial loss.

The longer a transaction remains in the blockchain, the more dependent transactions will need to be reverted due to their interdependencies.

Furthermore, the potential to submit a Proof of Realisation at the host chain and reorganize the rollup necessitates that the bridge smart contract must await the finalization of the Fork Choice Rule (FCR) before accepting messages from the rollup to the host chain (e.g., a withdrawal). The FCR either becomes probabilistic or with an optimistic window. In the first case, the longer a block remains on the chain, the lower the probability that a Proof of Realisation will be submitted for the block and thus the higher the finalization certainty. In the second case, that the optimistic window is a fixed amount of time, after which the block is considered finalized and no Proof of Realisation will be accepted for a transaction that is included in that block.

Of course, the FCR approach has lower overhead to the rollup Node, as it doesn't need to run the Assertion Enforcer to check for Invalidating transactions. The rollup only needs to observe the submission of Proofs of Realisation and then the FCR will handle the removal of the invalidating transasction and any other cleanup.

5.3.2 Enshrining the modified STF

If the blockchain node runs the Assertion Enforcer, it instead modifies the STF of the node, as certain state transitions are considered invalid according to the assertions.

In that case, a hard fork on the rollup is needed to upgrade the smart contracts, but the Proof of Realisation now functions similar to a fraud proof [1]. A submission of this proof will **not** trigger a reorg, but will only slash the rollup Node that the block with the Invalidating transaction. The reason is that since the rest of the rollup Nodes operate as Assertion Enforcers, they will not consider the block as valid to begin with, and they will not build on top of it.

Thus, the modification of the STF results in a design that offers superb UX, since there are no reorgs and incorrect state transitions are simply not observed by the rest of the rollup Nodes.

5.3.3 Enshrining is difficult

Enshrining the Credible Layer on the rollup architecture requires the modification of the rollup smart contracts and node software. This makes the design rather impractical for existing rollups, which naturally are risk averse when it comes to their most critical infrastructure.

That leaves the design to be implemented on a new rollup architecture, which forks one of the existing rollup stacks (OP Stack [24], Arbitrum Orbit [3], etc.) and implements the changes in the FCR or STF. The adoption of such a design will be challenging, as it will either depend on existing rollups to adopt the forked software or depend on nascent new rollups. The first case is rather unlikely, as the rollup node and smart contract code is considered critical infrastructure and every line of new code is a possible attack vector. The second case is also an uphill battle, as users don't follow the best technology, but rather network effects and inertia.

On the other hand, the proposed design for the Credible Layer that has been described in this paper is modular and can be implemented as a separate module, that can run alongside the existing rollup node as a middleware, without requiring any change to either the rollup node or the smart contracts. That means that it's much easier to be adopted by existing rollups as an extra value proposition for their users. It enables the Credible Layer to go where the users are and also require minimum integration effort from both the rollup Nodes (Assertion Enforcers) and the dApps (Assertion Adopters).

5.4 Enshrining Hack Prevention in L1s

What we described above can very well be applied to the STF of an L1, like Ethereum. Although such a design seems rather impractical, it is actually the very first draft design [34] of what became later the Credible Layer.

First, let's make a quick mention to the FCR rules of Ethereum. It employs two primary fork-choice rules, LMD-GHOST and Casper FFG, with the latter being considered essential to the protocol.

The first rule involves attesters and proposers utilizing a modified LMD-GHOST algorithm to identify the canonical chain tip. Block proposers are expected to build upon the tip selected by this rule, while attesters should only endorse attestations confirming their perceived tip. The second rule mandates that nodes implement the Casper FFG mechanism for block finalization. Once a node's view of Casper FFG has confirmed a block, that node must reject any chain reorganization attempting to remove said block.

Given this design, we can propose a modified version of the Credible Layer, where proposers are slashed if they vote (during the Casper FFG mechanism [9]) for a block that includes a transaction which invalides some assertion of a dApp. If said block fails to receive votes from 2/3 of the quorum, then it will never be finalized, and instead another block will finalize in it's stead.

Enshrining the Credible Layer in this design effectively means that a quorum of the proposers will operate as Assertion Enforcers, which will modify their STF, which in turn will consider Invalidating transactions as invalid, and thus their FCR will not vote for blocks containing those transactions.

The proposed system here functions, effectively, as an incentive mechanism for consensus attacks on the network, attacking any block (by not voting) which doesn't adhere to the out-of-band rules introduced by the Credible Layer. The attack leverages the fact that Ethereum doesn't have Single Slot Finality [22] (such as Tendermint [30]), enabling reorging "out" the "malicious" blocks. Naturally, the introduction of SSF would render this design obsolete, as blocks would finalize as soon as they are proposed.

Although unrealistic, it serves as another interesting example of how the STF and FCR can be modified and extended, outside of the protocol, affecting the behavior of the protocol itself. Effectively, if such a majority adopted the design, we would have a hard-fork of the protocol, as any block with an Invalidating transaction would never finalise, pushing the rest of the network to also operate Assertion Enforcers.

5.5 Enshrining Provable Off-Chain Computations

Now, we want to explore the Credible Layer design in the context provable off-chain computations. We use the term "Enshrining" here a bit more loosely, to describe that the underlying Node is aware of these computations and uses them either via the filtering or enshrined approach.

The dApps don't need to verify the correct execution of every single off-chain computation (state transition of a transaction), but can instead observe the outcome of these compuatations and challenge any incorrect execution. In our case, the incorrect outcome would be the inclusion of an Invalidating transaction, which puts the Assertion Adopter in a state where some posted assertions are invalidated. Once observed, anyone can submit a valid *Proof of Realisation* and thus economically penalize the Assertion Enforcer.

The same design approach can instead be used to build off-chain applications all-together. The off-chain computations have their own state and advance from user interactions, and if any user identifies an incorrect state transition, they can submit a proof and the chain will fork accordingly. InfinityVM [46] is using such a design approach to build off-chain applications.

5.5.1 Counter Factual Applications

Amusingly, one could potentially build complete applications on the Credible Layer by leveraging assertions. As per the *Checks-Effects-Interact* pattern [14] for writing secure smart contracts, code can be broken up into three groups:

1. **Check**: Code that checks conditions around the interaction (e.g check the withdraw amount is less or equal than the user balance)
2. **Effect**: Code that changes the state of the smart contract (e.g update user balance)
3. **Interact**: Code that interacts with other smart contracts (e.g transfer ERC20 tokens)

A dApp could implement the **Checks** as assertions instead, leaving the smart contract to only handle effects and interactions. We call this approach *Counter*

Factual Applications because the assertions dictate what the user can't do, and the smart contract only handles the allowed interactions.

5.6 Assertion blocks

Finally, the Credible Layer can be conceptualized as a blockchain architecture superimposed on the base layer blockchain:

1. Each block in the base layer blockchain contains a finite set of transactions.
2. A transaction can interact with an arbitrary number of smart contracts, a subset of which are designated as Assertion Adopter contracts.
3. Each Assertion Adopter contract is associated with a set of assertions.
4. During the execution of a transaction within a block, it may interact with multiple Assertion Adopter contracts, potentially triggering numerous assertions.
5. The Assertion Enforcer, constrained by computational resources, imposes an upper bound on the number of assertions that can be executed for a single transaction.
6. Consequently, each assertion can be viewed as generating a block of assertions on the Credible Layer.

5.7 Conclusion

In conclusion, the Credible Layer offers a modular and flexible approach to enhancing blockchain security and functionality. By integrating as a separate module that can run alongside existing rollup nodes, it minimizes the integration effort required from both rollup Nodes (Assertion Adopters) and dApps (Assertion Adopters). The design can be applied to various contexts, including enshrining hack prevention mechanisms directly into Layer 1 (L1) protocols like Ethereum, where modifications to the State Transition Function (STF) and Fork Choice Rule (FCR) can enforce security measures at the consensus level. Additionally, the Credible Layer can support provable off-chain computations, allowing dApps to challenge incorrect state transitions and economically penalize Assertion Enforcers through Proof of Realisation. The concept of Counter Factual Applications further demonstrates the versatility of the Credible Layer by enabling dApps to implement checks as assertions, simplifying smart contract logic. Finally, the idea of assertion blocks introduces a new layer of blockchain architecture that operates in parallel with the base layer, ensuring efficient and scalable transaction validation.

6 Future Work

Future work on the Phylax Credible Layer will focus on several key areas to enhance its functionality, scalability, and adoption. These include:

- Decentralize Assertion Data Availability and Fee calculation.
- Formalize the security model, and quantify the economics of the mechanism, such as rewards, security budget and penalties.
- Enhance capital efficiency by integrating with restaking protocols, developing security bond markets and on-chain insurance programs.
- Expand the PhEVM capabilities to access historical states as also support cross-chain assertions. Implement optimizations for the execution of assertions within the PhEVM and ensure that the overhead on the network remains low.
- Implement designs for networks with decentralized block-building, including application-specific sequencing.
- Expand the middleware design from the OP-stack and support all rollup architectures, such as Arbitrum Nitro.
- Identify products and use-cases that can be built on top of the Credible Layer and support the development of a vibrant ecosystem.

These efforts will contribute to making the Credible Layer a more robust, versatile, and widely adopted security solution for the blockchain ecosystem.

If you are interested in working on these topics, please reach out to us at odysseas@phylax.watch.

7 Conclusion

The Phylax Credible Layer introduces a novel approach to blockchain security, addressing challenges in preventing and mitigating hacks within decentralized applications. It combines on-chain incentive alignment mechanisms with off-chain software for detecting improper state transitions.

Key characteristics of the Credible Layer

1. State Oracle: Defines protocol-compliant state transitions.
2. Assertion Enforcement Mechanism: Incentivizes network actors to act as Assertion Enforcers.
3. Proof of Possibility (PoP) and Proof of Realization (PoR): Cryptographic proofs for verifying realized or possible invalid states.

4. Economic Incentives: Aligns interests of participants, creating a sustainable security ecosystem.

5. Modular Architecture: Allows integration with existing rollup solutions and potential expansion to Layer 1 networks.

The paper explores potential extensions, such as integration with restaking protocols and security bond markets, to enhance capital efficiency and security.

While the Credible Layer addresses many existing security challenges, it also introduces new considerations, which we plan to address in future work.

As the blockchain industry evolves, the Phylax Credible Layer offers an adaptable framework for proactive security measures and incentivized ethical hacking.

8 Acknowledgements

We would like to thank the following people for reviewing the paper and providing valuable feedback: Brock Elmore, James Parillo, Fred Luehrs, Will Price, Nick Almond, Piotr Szlachciak, and Tarun Chitra.

A Appendix

A.1 Assertion Accuracy

In this section, we will explore what makes an assertion accurate, in the context of security. In other words, what are the assertions that the mechanism should give incentives for in order to be maximally effective. For example, an assertion that is always true is not very useful, as it does not provide any security guarantees, but just wastes resources. Moreover, we will explore the notion of a vulnerability set and who underwrites the accuracy of the assertion. The latter showcases a few design areas that we can explore in the future and expand the mechanism.

Definition A.1 (Vulnerability Set)**.** The set of all existing vulnerabilities in a smart contract, which is typically unknown or only partially known at any given time. This set may include:

- Known vulnerabilities that have been identified but not yet exploited or patched
- Unknown vulnerabilities that have not yet been discovered
- Potential vulnerabilities arising from the interaction of the contract with other contracts or external systems

The vulnerability set is dynamic and may change over time as new vulnerabilities are discovered or existing ones are patched.

Definition A.2 (Accurate Assertion)**.** An accurate assertion $a \in A$ is a boolean function $a : S \to \{\text{true}, \text{false}\}$ such that:

1. $\exists v \in V : a(v) = \text{false}$
2. $\forall s \in S \setminus V : a(s) = \text{true}$

where $V \subset S$ is the vulnerability set of a Assertion Adopter, and S is the set of all possible states.

In other words, an accurate assertion is a function that:

1. Correctly identifies at least one state in the vulnerability set by evaluating to false.
2. Correctly evaluates to true for all states that are not in the vulnerability set.

This definition ensures that an accurate assertion captures vulnerabilities from the vulnerability set without producing false positives for non-vulnerable states.

Definition A.3 (Inaccurate Assertion)**.** An inaccurate assertion $a' \in A$ is a boolean function $a' : S \to \{\text{true}, \text{false}\}$ that exhibits at least one of the following properties:

1. $\exists s \in S \setminus V : a'(s) = \text{false}$ (False Positive)
2. $\forall v \in V : a'(v) = \text{true}$ (False Negative)
3. $\forall s \in S : a'(s) = \text{true}$ (Always True)
4. $\forall s \in S : a'(s) = \text{false}$ (Always False)
5. $\exists s \in S : s$ is an unreachable state in the Assertion Adopter's code and $a'(s) = \text{false}$ (Unreachable State)

where $V \subset S$ is the vulnerability set of a Assertion Adopter, and S is the set of all possible states.

A.1.1 Importance of Accurate Assertions

Accurate assertions are essential because:

- They correctly identify vulnerabilities without producing false positives or negatives.
- They contribute to the security of the system by ensuring that all states in the vulnerability set can be detected.
- They optimize resource utilization by avoiding unnecessary computations for irrelevant or impossible states.

Similar to how Ethereum uses gas costs to deter spam, the Credible Layer should incorporate mechanisms to discourage the creation of inaccurate assertions. This ensures that computational resources are efficiently utilized and that the system maintains a high level of security and reliability.

A.1.2 Who underwrites the Assertion Accuracy

In the model described above, the Assertion Adopter underwrites the accuracy of the assertion by whitelisting particular Assertions or Assertion Submitters.

This mechanism can be enhanced by allowing the Assertion Submitter to underwrite the accuracy of the assertion, staking a specified amount of funds that will be slashed if an attack succeeds.

Of course, the successful attack needs to be defined, as by definition it won't result to any assertion being invalidated from that particular Assertion Submitter.

Possible research directions include:

1. **Governance mechanism**: A group of users can vote to signal that a Assertion Adopter was "hacked" and the assertion group was incomplete, either through the governance token of the Assertion Adopter (if it exists) or some other protocol like UMA.

2. **Assertion Groups quorum**: If there are multiple Assertion Groups and only a subset of them is invalidated, if a quorum is reached, the Assertion Submitters of the non-invalidated Assertion Groups will be slashed.

Essentially, this is a form of assumption of liability, whereby the Assertion Submitter is the one that is liable for the assertion and the potential consequences of not accurately capturing the vulnerability. A form of insurance for the Assertion Adopter, provided by the Assertion Submitter.

A.2 Fees Sketch

In this section we will sketch a few ideas on how we can price the assertion execution and definition.

A.2.1 Baseline Assertion Fee

Consider a block with n transactions, and for some transactions, there are m_i assertions. Let $G_{i,j,k}$ be the gas spent on the k-th assertion of the j-th Assertion Adopter in the i-th transaction. The total gas T spent on all assertions for the block is:

$$T = \sum_{i=1}^{n} \sum_{j=1}^{N_i} \sum_{k=1}^{m_{i,j}} G_{i,j,k}$$

where N_i is the number of Assertion Adopters interacted with in the i-th transaction, and $m_{i,j}$ is the number of assertions for the j-th Assertion Adopter in the i-th transaction.

Assertion Execution Fee To calculate the assertion execution fee M for a block, we introduce a scaling factor α and a baseline fee β. The assertion execution fee is then calculated as:

$$M = \alpha \cdot T + \beta$$

Here, α adjusts the impact of the total gas spent on the assertion execution fee, and β is the baseline fee that represents fixed operational costs.

For a total of k blocks, the overall assertion execution fee O incorporates both the scaled total gas spent on assertions and the baseline fees across all blocks. If T_l represents the total gas spent on assertions in the l-th block, the overall assertion execution fee is:

$$O = \sum_{l=1}^{k} (\alpha \cdot T_l + \beta) = \alpha \cdot \sum_{l=1}^{k} T_l + k \cdot \beta$$

This formula calculates the overall assertion execution fee by scaling the total gas spent across all blocks and adding the baseline fee for each block. The term $\sum_{l=1}^{k} T_l$ represents the total gas spent on assertions across all blocks, α is the scaling factor, β is the baseline fee, and k is the number of blocks.

Assertion Definition Fee The assertion definition fee employs an inverse scaling factor α^{-1} to incentivize the submitted assertions to use minimal gas, rewarding efficiency with higher fees relative to gas savings. This fee is calculated as:

$$A = \alpha^{-1} \cdot T + \beta$$

where T is the total gas used, and β represents a baseline fee. This approach encourages the Assertion Submitter to optimize operations, benefiting the dapp by reducing unnecessary computational expenses on the Assertion Enforcer.

The payment of the Assertion Submitter could be very well defined out-of-protocol, in the interest of simplicity. The mechanism enables anyone to submit assertions, and it is expected that Assertion Adopters will provide out-of-protocol incentives for people to do so, such as through a contractual agreement.

A continuous payment scheme that is enforced by the protocol has a very particular benefit, in terms of making security less capital intensive. Instead of the Assertion Adopter having to allocate a lump sum of funds upfront, it can simply pay as it goes. In the case that the Assertion Adopter behave maliciously, for example by re-submitting an assertion that was first submitted by another Assertion Submitter, it signals to the community that the Assertion Adopter is not reliable and should be avoided. It's worth mentioning that the capital intensity of security, particularly audits, ranging in the several hundreds of thousands of dollars, is a very real issue that a lot of nascent dApps struggle with.

A.2.2 Dynamic Assertion Fee

Of course, the vigilant reader will already have observed that given the limited resources of the Assertion Enforcer, the above is not exactly accurate.

Some transactions might invoke multiple assertions at the same time, and some might not. Not dissimilar to how gas works in blockchains, not all blocks are created equal and competition for inclusion in a "hot" block means that the economical cost will rise.

Dynamic Assertion fees seem important if we consider the possibility of *not* running some assertions in the face of congestion. Since that would open up the system to attacks, we consider such an approach ill-advised. In that case, dynamic fees seem less important, as there will never be congestion. The Assertion Enforcer will always be able to execute all the assertions even in the worse-case scenario.

Obviously, we don't make such a statement lightly, and although we make a first attempt at illustrating the problem in the following sub-section, we consider this issue an open question that requires further research, experimentation and formalisation.

A.2.3 Fees as a Spam Protection Mechanism

The way the Credible Layer handles Spam Protection is through the use of a gas metering system for Assertions. When Assertions are activated, the Assertion Enforcers are entitled to rewards from the security budgets of the Assertion Adopters. Assertion Enforcers, as discussed, are treated optimistically by the system, as far as assertion execution is concerned.

They are *expected* to execute assertions after every transaction and ensure that no Invalidating transaction lands on the network. In the same manner, they can claim rewards optimistically, without having to prove that indeed they ran the assertions. If a Assertion Adopter's security budget is exhausted, then it's assertions are no longer considered active and thus the Assertion Enforcer is allowed to not execute them, without the fear of a slashing event. Assertion Adopters are expected to keep track of their security budget and deposit new funds to make sure their assertions remain active.

By incurring an economical cost to the Assertion Adopters for activating assertions, it protects the system and more specifically the Assertion Enforcers from malicious Assertion Adopters that simply want to waste it's resources.

Spam can be expressed as two types of attacks:

1. Resource Starvation Attack: Invoke as many Assertions as possible during a single state transition, with the intent to starve the Assertion Enforcer from resources and create a bottleneck in the transaction inclusion process (liveness) of the Network

2. Security Budget Starvation Attack: Invoke as many assertions as possible, not with the intention of interacting with the Assertion Adopter in order to derive some value, but to waste their security budget

To illustrate the point, let's see the worse case scenario where a user tries to attack the protocol by invoking as many assertions as possible at the same time.

To establish a theoretical upper bound on the number of assertions executed per block, we consider the following constraints and assumptions:

1. Each block on the base layer has a fixed gas limit.
2. We assume the existence of a contract designed to interact with as many Assertion Adopters smart contracts as possible within a single transaction.
3. Assertions are triggered when a Assertion Adopter's state changes, which occurs when:
   (a) Any storage variable is updated
   (b) The contract's balance changes
4. We assume the attack contract spends the minimum amount of gas required to interact with an Assertion Adopterand invoke it's assertions

A.2.4 Theoretical bound of executed Assertions

This is the absolute worse-case scenario, as sending ETH is cheaper than any operation that includes changing the storage of a contract.

Let:

$$\begin{aligned} G_b &= \text{Gas limit per block} \\ G_t &= \text{Gas cost of a minimum value transfer} = 21000 \text{ (base transaction cost)} \\ G_c &= \text{Gas cost of a contract call} = 700 \text{ (minimum contract call cost)} \\ N_{max} &= \text{Maximum number of Assertion Adopter contracts affected per block} \end{aligned}$$

For EVM networks, we need to consider both the transfer cost and the contract call cost:

$$N_{max} = \left\lfloor \frac{G_b - G_t}{G_c} \right\rfloor = \left\lfloor \frac{G_b - 21000}{700} \right\rfloor$$

Consequently, the theoretical upper bound on the number of assertions executed per block is:

$$A_{max} = N_{max} \times A_{aa} \tag{6}$$

Where A_{aa} is the number of assertions associated with each Assertion Adopter contract.

A.2.5 Upper Bound of Executed Assertions on Base Network

Let's take the Base Network [6] as a case-study for our analysis.

At the time of writing, the following properties hold true for Base:

$$\begin{aligned} G_b &= 120,000,000 \text{ (Gas limit for Base)} \\ G_t &= 21,000 \text{ (Base transaction cost)} \\ G_c &= 700 \text{ (Minimum contract call cost)} \end{aligned}$$

We can calculate the maximum number of Assertion Adopter contracts affected per block:

$$N_{max} = \left\lfloor \frac{G_b - G_t}{G_c} \right\rfloor = \left\lfloor \frac{120,000,000 - 21,000}{700} \right\rfloor = 171,398$$

Therefore, the theoretical upper bound on the number of assertions executed per block is:

$$A_{max} = 171,398 \times A_{aa}$$

Where A_{aa} is the number of assertions associated with each Assertion Adopter contract.

Now, let's consider a gas limit for assertions for each Assertion Adopter contract:

$$A_{aa} = 200,000 \text{ (Gas limit for assertions for each Assertion Adopter contract)}$$

Consequently, the theoretical upper bound on the total gas consumed by assertions for a single block is:

$$A_{max} = 171,398 \times 200,000 = 34,279,600,000 \text{ gas} = 34.2796 \text{ Mgas}$$

A.2.6 Resource Starvation Attack on Assertion Enforcers

Attack Definition The Resource Starvation Attack is a liveness attack targeting the transaction processing capabilities of a blockchain network. The attack exploits the system's assertion mechanism by invoking an excessive number of assertions during a single state transition.

Attack Mechanism Let S be the set of all possible system states, T be the set of all possible transactions, and A be the set of all possible assertions. We define:

$$\begin{aligned} s_t \in S &: \text{The system state at time } t \\ \tau \in T &: \text{A transaction} \\ a_i \in A &: \text{An individual assertion} \\ AE : S \times T \times 2^A \to S &: \text{The Assertion Enforcer function} \end{aligned} \tag{7}$$

The attacker constructs a malicious transaction τ_m such that:

$$\tau_m = \{a_1, a_2, ..., a_n\} \tag{8}$$

Where n is extremely large, approaching or exceeding the system's processing capacity.

Attack Objective The primary goal of this attack is to impede the liveness of the network by:

1. Overwhelming the Assertion Enforcer:

$$AE(s_t, \tau_m, \{a_1, a_2, ..., a_n\}) \approx \text{timeout} \tag{9}$$

2. Creating a bottleneck in transaction processing:

$$\forall \tau \in T, \text{ProcessingTime}(\tau) \gg \text{NormalProcessingTime}(\tau) \tag{10}$$

Impact on Network Liveness Let $L(t)$ be a function representing the liveness of the network at time t. The attack aims to significantly reduce $L(t)$:

$$L(t + \Delta t) \ll L(t) \tag{11}$$

Where Δt is the duration of the attack.

Potential Consequences

1. Increased transaction latency
2. Reduced network throughput
3. Backlog of pending transactions
4. Temporary denial of service

Mitigation Strategies To counter the Resource Starvation Attack, we propose and analyze several mitigation strategies. Let τ_m be a malicious transaction designed to overwhelm the Assertion Enforcer (AE). We define the set of all assertions triggered by τ_m as $A = \{a_1, a_2, ..., a_n\}$.

Potential Countermeasures

1. Imposing limits on assertions per transaction:

$$|A| \leq k, \text{ where } k \text{ is a predefined constant} \tag{12}$$

2. Employing dynamic pricing mechanisms: Let $C(a_i)$ be the cost of executing assertion a_i. We define a dynamic pricing function $P(A)$:

$$P(A) = f\left(\sum_{i=1}^{n} C(a_i), |A|, t\right) \tag{13}$$

   Where t is the current network state, and f is a monotonically increasing function.

Decision Framework for the Assertion Enforcer When the AE encounters τ_m, it must choose from the following options:

1. Refuse to process the transaction:

$$\text{If } |A| > k \text{ or } \sum_{i=1}^{n} R(a_i) > R_{max}, \text{ then Reject}(\tau_m) \tag{14}$$

2. Process a subset of assertions:

$$A' \subset A, \text{ such that } |A'| \le k \text{ and } \sum_{a_i \in A'} R(a_i) \le R_{max} \tag{15}$$

3. Allow bounded latency: Let $L(\tau)$ be the processing latency of transaction τ, and L_{max} be the maximum acceptable latency.

$$\text{Process}(\tau_m) \text{ iff } L(\tau_m) \le L_{max} \tag{16}$$

Analysis of Options Option 2 (processing a subset of assertions) is eliminated because it contradicts the mechanism's desiderata of ensuring that all assertions are executed (Security). We focus on Options 1 and 3:

- Option 1 (Rejection): Viable if we can define a threshold k^* such that:

$$P(|A| > k^* | \tau \text{ is benign}) \approx 0 \tag{17}$$

  Where $P(\cdot)$ denotes probability.

- Option 3 (Bounded Latency): Viable if we can establish a latency limit L_{max} such that:

$$P(L(\tau) > L_{max} | \tau \text{ is benign}) \approx 0 \tag{18}$$

  And:

$$U(L_{max}) \approx U(L_{normal}) \tag{19}$$

  Where $U(\cdot)$ represents user experience quality, and L_{normal} is the typical latency under normal operation.

By carefully tuning the parameters k^*, R_{max}, and L_{max}, it seems that this attack can be mitigated. Additional restrictions can be implemented to further mitigate the risk of an attack, such as limiting the number of assertions that can be defined per contract (AA), as also setting a hard limit on the gas cost of an assertion execution. If this restrictions are clearly communicated and defined in the protocol, the Assertion Adopter is expected to provide appropriate assertions that don't overstep these limits.

Assertion Triggers: A Fine-Grained Approach To mitigate the worst-case scenario described in the Resource Starvation Attack, we propose a more nuanced mechanism for controlling assertion execution conditions. This approach enables fine-grained control over when assertions are triggered, potentially alleviating the computational burden on the Assertion Enforcer.

Let A be the set of all assertions, and for any assertion $a \in A$, we define the following trigger conditions:

$$T(a) = \begin{cases} 1, & \text{if trigger condition is met} \\ 0, & \text{otherwise} \end{cases} \tag{20}$$

We introduce three specific trigger types:

1. $T_s(a)$: Trigger when any storage slot changes
2. $T_b(a)$: Trigger when ether balance changes
3. $T_c(a)$: Trigger on both storage and balance changes

Formally, we can express these triggers as follows:

$$\begin{aligned} T_s(a) &= \begin{cases} 1, & \text{if } \exists s \in S : s_{t+1} \neq s_t \\ 0, & \text{otherwise} \end{cases} \\ T_b(a) &= \begin{cases} 1, & \text{if } b_{t+1} \neq b_t \\ 0, & \text{otherwise} \end{cases} \\ T_c(a) &= \max(T_s(a), T_b(a)) \end{aligned} \tag{21}$$

Where S is the set of storage slots, s_t is the state of a storage slot at time t, and b_t is the ether balance at time t.

To further refine the system's resilience against potential attacks, we introduce a gas pricing mechanism that differentiates between trigger types:

$$G(a) = \begin{cases} g_s, & \text{if } T_s(a) = 1 \\ g_b, & \text{if } T_b(a) = 1 \\ g_c, & \text{if } T_c(a) = 1 \end{cases} \tag{22}$$

Where g_s, g_b, and g_c are the gas costs for storage, balance, and combined triggers respectively, with $g_b > g_s$ to reflect the higher computational cost of ether transfers.

To prevent denial-of-service attacks, we implement a gas limit L for contracts:

$$\sum_{a \in A'} G(a) \leq L \tag{23}$$

Where $A' \subseteq A$ is the subset of assertions triggered by a single transaction.

This gas limit ensures that no transaction, malicious or otherwise, can trigger an excessive number of assertions that would overwhelm the Assertion Enforcer's parallel processing capacity.

By carefully tuning the parameters g_s, g_b, g_c, and L, we can create a system where:

$$P(\text{DOS}) \approx 0 \tag{24}$$

Where $P(\text{DOS})$ represents the probability of a successful denial-of-service attack.

It's worth mentioning that the assertion triggers can be further expanded to include more fine-grained conditions, such as specific storage slot changes or even specific function calls. By increasing the cost of the assertion calls the more generic they are, we incentivise Assertion Adopters to be very intentional with what code paths should trigger which assertions. This specificity enables the system to be more robust to attacks which intend to either waste computational resources of the Assertion Enforcers or waste security budget of the Assertion Adopters.

A.2.7 Security Budget Starvation on Assertion Adopters

The Security Budget Starvation attack is a strategy where an attacker attempts to deplete the security budget of Assertion Adopters by triggering excessive assertion executions. This attack exploits the mechanism where Assertion Adopters pay Assertion Enforcers for executing assertions, potentially leading to premature exhaustion of Assertion Adopters' security budgets.

Attack Mechanism Let B_p be the security budget of a Assertion Adopter p, and C_a be the cost of executing an assertion a. An attacker generates a series of transactions $T = \{t_1, t_2, ..., t_n\}$ that trigger assertions $A = \{a_1, a_2, ..., a_m\}$. The attack succeeds if:

$$\sum_{i=1}^{m} C_{a_i} \geq B_p \tag{25}$$

Inherent Economic Disincentive It's crucial to recognize that this attack inherently carries an economic cost for the attacker. Each transaction that triggers assertions requires the attacker to pay gas fees on the underlying layer. This built-in economic penalty serves as a first line of defense against frivolous or malicious triggering of assertions.

Let G_t be the gas cost of a transaction t. The total cost to the attacker for a series of transactions T is:

$$C_{attack} = \sum_{t \in T} G_t \tag{26}$$

For the attack to be economically viable, the attacker must perceive a benefit greater than this cost:

$$Benefit_{attack} > C_{attack} \tag{27}$$

Since there is no direct benefit to the attack, one must explore second-order effects to possibly model the benefit of the attack. For example, it could be a hacker that wants to exhaust the security budget over a long period of time in an effort to make the protocol remove assertions and thus open a window of opportunity to exploit a vulnerability. It could also be a competing protocol that simply wants to inflict financial damage to their competitors.

The reality is that this type of attack is inherent to the mechanism we have described and tackling it a the protocol level seems unlikely, unless we change the mechanism itself.

Currently, there are two approaches that are being explored:

1. Allow this cost to be externalised to the user at the Assertion Adopter level. For example, Assertion Adopters could add a tax by modifying the smart contract code, effectively externalising the cost.

2. Enshrine the payment of assertions at the transaction level, much like every transaction pays the base layer in gas. New innovations in the area of smart accounts would enable such an approach without requiring any change at the base layer, but could add friction to the user experience by requiring them to acquire the Credible Layer gas token ahead of interacting with a Assertion Adopter.

A.3 Credible Layer Mechanism as a Restaking Module

In the cryptocurrency ecosystem, restaking has emerged as an innovative approach to enhance capital efficiency. This concept enables participants to utilize their staked tokens across multiple networks concurrently, effectively securing various protocols with the same assets. While this strategy can potentially yield additional rewards for users who secure multiple protocols, it also introduces a higher level of risk in terms of potential slashing.

In Ethereum, restaking was popularised and introduced by EigenLayer, where protocols can be designed as "Actively Validated Services" (AVS) to leverage the staked ETH to secure the network via EigenLayer. Effectively, a subset of the Ethereum Validators can choose to use their staked ETH to secure the protocol, by delegating their restaked ETH to the Operators, who opt in to provide various services (AVSs) built on top of EigenLayer.

Although we will be using EigenLayer as restaking protocol to sketch the AVS extension of the mechanism, the same principles can be applied to any restaking protocol, such as Symbiotic. In fact, one could extend the mechanism in such a way where both integrations co-exist.

The design of the Credible Layer remains the same, apart from the fact that the Assertion Enforcers are now Operators of the Credible Layer AVS. Instead of the Assertion Enforcer bonding a security budget, they can instead bond the restaked ETH which validators have delegated to them. In the case of a PoNM submission, the slashing functionality of the Credible Layer calls the Eigenlayer protocol to slash the validators who delegated their restaked ETH to the operator.

A.3.1 Assertion Enforcer as an Operator

Conversely, the Assertion Enforcer can repurpose a percentage of the security fees that they get from the Credible Layer to provide yield to the validators who have delegated their restaked ETH to them.

The primary benefit of this approach is that now the Assertion Enforcer doesn't have to bond and lock a significant percentage of value, but instead they can use the delegated ETH and provide a percentage of their revenue to the validators.

Assertion Enforcer	**Validators**
Uses delegated ETH	Delegate restaked ETH to Assertion Enforcer
Provides a percentage of revenue as yield to validators	Receive yield in exchange for capital risk
Responsible for honest operation to avoid slashing	Trust Assertion Enforcer to operate honestly

Table 2: Relationship between Assertion Enforcers and Validators in the Credible Layer AVS

This arrangement creates a mutually beneficial relationship where Assertion Enforcers can operate with less capital lock-up, while validators can earn additional yield on their staked ETH.

A.4 Security Bond Markets

The restaking module described in the previous subsection can be generalized into a Security Bond Market, wherein Assertion Enforcers can issue bonds to raise capital. These bonds can be purchased by any individual or institution that seeks to invest in the future revenue of the Assertion Enforcer or the Assertion Submitter.

Specifically, the mechanism identifies two parties with ongoing and verifiable income streams, namely the *Assertion Enforcers* and *Assertion Submitters*.

These parties can issue tokens that grant the owner a claim on their future revenue, effectively functioning as bonds that provide yield to the owner through coupons.

A reputable Assertion Enforcer, such as the Base Sequencer, will be able to issue bonds at a relatively low cost due to the low likelihood of malicious behavior. Conversely, a relatively new Block Builder or network may need to offer higher yields to attract bond buyers.

The advantages of such markets are two-fold:

1. **Capital Efficiency**: Assertion Enforcers can participate in the mechanism with less of their own capital, as they can attract external capital through the Security Bond Market.

2. **Mechanism Security**: As discussed in Section 2.7, the bonded security budget of the Assertion Enforcer will realistically never be sufficient to offset the potential bribery value in purely economic terms, as hacks typically involve sums in the tens of millions. With Security Bonds, the system can increase bond requirements, allowing the Assertion Enforcer to leverage both owned and external capital.

Furthermore, this concept can be extended to the Assertion Adopter, which can attract capital for their own security budgets to pay for security fees and bounties. In this scenario, the yield would be provided in the form of protocol-owned tokens (e.g., Governance tokens), and the bond buyers would be betting on the Assertion Adopter **not** needing to pay out bounties.

A.5 Security Bond Markets as Insurance

An extension to the previous discussion on security bond markets pertains to the concept of insurance.

Insurance is a complex subject, as on-chain insurance projects have traditionally struggled to amass sufficient capital to be effective. There are a few crypto-native insurance companies, but they are explicitly not automated or natively on-chain, as they require investigation before honoring their insurance contracts. This requirement is understandable, as they must rule out situations where the protocol was overly negligent to security best practices or where there the attack was carried out from employees or generally the company itself.

Nevertheless, we can utilize bond markets as a medium to potentially design a realistic on-chain insurance program for end-users.

The construct is as follows:

1. An end-user of a Assertion Adopter can become **Insured** and opt into an insurance policy by paying a form of tax while interacting with the Assertion Adopter.

2. Any individual can operate as an **Insurer** by staking funds into the insurance fund, thereby earning a percentage of the reward from it. The longer they lock their funds, the higher the yield they are entitled to.

3. All taxes collected from the insured end-users are distributed to the insurance fund stakers.
4. The insurance policy is tied to specific assertions of the Assertion Adopter, meaning that the insurance only covers slashing events related to the submission of PoNMs that occur as a result of these assertions.
5. In the event of a successful PoNM submission, the insurance fund stakers lose their staked funds, which are then distributed to the insured users.

The primary benefit of this construct is that it creates a market-driven mechanism to ensure that the insurance fund is always sufficiently large to cover the payouts in the event of a slashing incident. **Insurers** are essentially betting on the integrity of the Assertion Enforcer to prevent any hacks. Due to the nature of the protocol, there is no ambiguity regarding the policy coverage or the associated risk, as all parties are known.

Users can become insured by paying a tax when interacting with dApps. Anyone can become an insurer by staking funds into the insurance pool, earning rewards from user taxes. Insurance policies are tied to specific assertions, ensuring clear coverage. In the event of a hack, proven cryptographically, insured users are compensated from the staked funds.

References

[1] John Adler and Mikerah. *Minimal Viable Merged Consensus.* Accessed: 2024-10-05. 2019. URL: `https://ethresear.ch/t/minimal-viable-merged-consensus/5617`.

[2] Arbitrum. *The Sequencer and Censorship Resistance.* Accessed: 2024-10-10. URL: `https://docs.arbitrum.io/how-arbitrum-works/sequencer`.

[3] *Arbitrum Orbit.* Accessed: 2024-10-05. URL: `https://arbitrum.io/orbit`.

[4] Georgia Avarikioti, Orfeas Stefanos Thyfronitis Litos, and Roger Wattenhofer. *Cerberus Channels: Incentivizing Watchtowers for Bitcoin.* Cryptology ePrint Archive, Paper 2019/1092. 2019. URL: `https://eprint.iacr.org/2019/1092`.

[5] *Axiom.* Accessed: 2024-10-05. URL: `https://axiom.xyz`.

[6] Base. *Base.* Accessed: 2024-10-07. URL: `https://base.org`.

[7] *Blast documentation.* Accessed: 2024-10-30. URL: `https://docs.blast.io/about-blast`.

[8] Jeremy Bruestle, Paul Gafni, and RiscZero Team. "Risc Zero zkVM: Scalable, Transparent Arguments of Risc-V Integrity". In: (2023). URL: `https://dev.risczero.com/proof-system-in-detail.pdf`.

[9] Vitalik Buterin and Virgil Griffith. “Casper the Friendly Finality Gadget”. In: *CoRR* abs/1710.09437 (2017). arXiv: `1710.09437`. URL: `http://arxiv.org/abs/1710.09437`.

[10] Jon Charbonneau. *Do Rollups Inherit Security?* Accessed: 2024-10-05. 2023. URL: `https://dba.xyz/do-rollups-inherit-security`.

[11] *Cream Finance Exploit – Oct 27, 2021-Detailed Analysis.* Accessed: 2024-10-30. 2022. URL: `https://www.immunebytes.com/blog/cream-finance-exploit-oct-27-2021-detailed-analysis/`.

[12] francesco d’amato and Mike Meuder. *Equivocation attacks in mev-boost and ePBS.* Accessed: 2024-10-10. 2024. URL: `https://ethresear.ch/t/equivocation-attacks-in-mev-boost-and-epbs/15338`.

[13] Philip Daian et al. *Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges.* 2019. arXiv: `1904.05234 [cs.CR]`. URL: `https://arxiv.org/abs/1904.05234`.

[14] Brock Elmore. *You’re Writing Require Statements Wrong.* Accessed: 2024-10-05. URL: `https://www.nascent.xyz/idea/youre-writing-require-statements-wrong`.

[15] *ERC-4337: Account Abstraction Using Alt Mempool.* Accessed: 2024-10-10. 2021. URL: `https://eips.ethereum.org/EIPS/eip-4337`.

[16] *Ethereum Block Builder Explorer.* Accessed: 2024-10-05. URL: `https://explorer.rated.network/builders?network=mainnet`.

[17] *Euler Hack.* Accessed: 2024-10-05. 2023. URL: `https://rekt.news/euler-rekt/`.

[18] Flashbots. *Builder API for the OP Stack.* Accessed: 2024-10-05. 2024. URL: `https://github.com/ethereum-optimism/specs/pull/116S`.

[19] Flashbots. *Flashbots MEV-Boost.* Accessed: 2024-10-07. URL: `https://docs.flashbots.net/flashbots-mev-boost/introduction`.

[20] Flashbots. *Relay Fundamentals.* Accessed: 2024-10-10. URL: `https://docs.flashbots.net/flashbots-mev-relay/mev-relay`.

[21] Flashbots. *Rollup Boost.* Accessed: 2024-10-30. 2024. URL: `https://github.com/flashbots/rollup-boost`.

[22] fradamt and Luca Zanolini. *A Simple Single Slot Finality Protocol.* Accessed: 2024-10-07. 2023. URL: `https://ethresear.ch/t/a-simple-single-slot-finality-protocol/14920`.

[23] *Fuzz.land.* Accessed: 2024-10-05. 2024. URL: `https://fuzz.land`.

[24] *Getting Started with the OP Stack.* Accessed: 2024-10-05. URL: `https://docs.optimism.io/stack/getting-started`.

[25] *Herodotus: Proving Ethereum’s State Using Storage Proofs on Starknet s.* Accessed: 2024-10-05. URL: `https://starkware.co/blog/proving-ethereums-state-on-starknet-with-herodotus/s`.

[26] Init4. *(Re)based Rollups*. Accessed: 2024-10-05. 2024. URL: https://blog.init4.technology/p/rebased-rollups.

[27] Georgios Konstantopoulos. *Intent-Based Architecture and Their Risks*. Accessed: 2024-10-05. URL: https://www.paradigm.xyz/2023/06/intents.

[28] Georgios Konstantopoulos and Dan Ronbinson. *Ethereum is a Dark Forest*. Accessed: 2024-10-05. 2024. URL: https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest.

[29] Satpal Singh Kushwaha et al. "Ethereum Smart Contract Analysis Tools: A Systematic Review". In: *IEEE Access* 10 (2022), pp. 57037–57062. DOI: 10.1109/ACCESS.2022.3169902.

[30] Jae Kwon. "Tendermint : Consensus without Mining". In: 2014. URL: https://api.semanticscholar.org/CorpusID:52061503.

[31] Celestia Labs. *Celestia*. Accessed: 2024-10-10. URL: https://celestia.org.

[32] Eclipse Labs. *A Deep Dive into Application-Specific Sequencing*. Accessed: 2024-10-10. URL: https://blog.eclipse.xyz/deep-dive-into-application-specific-sequencing.

[33] Eigen Labs. *EigenDA*. Accessed: 2024-10-10. URL: https://www.eigenda.xyz/.

[34] Odysseus Lamtzidis. *Blockchain Monitoring is MEV, always was*. Accessed: 2024-10-05. 2023. URL: https://odyslam.com/blog/monitoring-is-mev/.

[35] Seun Lanlege. *Merkle Mountain Range*. Accessed: 2024-10-05. 2023. URL: https://research.polytope.technology/merkle-mountain-range-multi-proofs.

[36] *Nomad Hack*. Accessed: 2024-10-05. 2022. URL: https://rekt.news/nomad-rekt/.

[37] Optimism. *Forced Transaction*. Accessed: 2024-10-10. URL: https://docs.optimism.io/stack/protocol/rollup/forced-transaction.

[38] Paradigm. *Revmc*. Accessed: 2024-10-10. URL: https://github.com/paradigmxyz/revmc.

[39] *Phalcon by BlockSec*. Accessed: 2024-10-05. 2024. URL: https://blocksec.com/blog/how-can-block-sec-phalcon-prevent-hacker-attacks-on-de-fi-protocols-by-using-front-running-techniques.

[40] James Prestwich. *Defining Rollup*. Accessed: 2024-10-05. 2023. URL: https://prestwich.substack.com/p/defining-rollup.

[41] *Proto-Danksharding*. Accessed: 2024-10-10. URL: https://www.eip4844.com/.

[42] *Radiant Capital Hack*. Accessed: 2024-10-30. 2024. URL: https://rekt.news/radiant-capital-rekt2/.

[43] *Roning Bridge Hack*. Accessed: 2024-10-05. 2024. URL: `https://rekt.news/ronin-rekt/`.

[44] Fabian Schär. *Blockchain Forks: A Formal Classification Framework and Persistency Analysis*. Feb. 2020. DOI: `10.13140/RG.2.2.27038.89928/1`.

[45] *Succinct SP1*. Accessed: 2024-10-05. URL: `https://github.com/succinctlabs/sp1`.

[46] The InfinityVM Team. *InfinityVM Litepaper: Enshrining Offchain Compute*. Accessed: 2024-10-07. 2024. URL: `https://infinityvm.xyz/infinityvm_litepaper.pdf`.

[47] Sergei Tikhomirov et al. "SmartCheck: Static Analysis of Ethereum Smart Contracts". In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 2018, pp. 9–16.

[48] Palina Tolmach. *Everything You Wanted to Know About Symbolic Execution for Ethereum Smart Contracts (But Were Afraid to Ask)*. Accessed: 2024-10-10. 2023. URL: `https://hackmd.io/@SaferMaker/EVM-Sym-Exec`.

[49] *Unlocking PBS with CL verification of block.coinbase*. Accessed: 2024-09-05. URL: `https://ethresear.ch/t/unlocking-pbs-with-cl-verification-of-block-coinbase-e-g-multi-tx-flashloans/17955/4`.

[50] Vac. *zkVM Explorations*. Accessed: 2024-10-10. 2024. URL: `https://vac.dev/rlog/zkVM-explorations/`.

[51] *Vyper Bug Hack*. Accessed: 2024-10-05. 2024. URL: `https://www.halborn.com/blog/post/explained-the-vyper-bug-hack-july-2023r`.

[52] Drew Van der Werff. *Beyond Extraction: The Role of Block Building in the Future of Ethereum*. Accessed: 2024-10-05. 2023. URL: `https://frontier.tech/beyond-extraction`.

[53] Gavin et al. Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[54] *Wormhole Hack*. Accessed: 2024-10-05. 2022. URL: `https://rekt.news/wormhole-rekt/`.

[55] Yuki Yuminaga. *A New Era of DeFi with App-Specific Sequencing*. Accessed: 2024-10-10. URL: `https://sorellalabs.xyz/writing/a-new-era-of-defi-with-ass`.