Thanks to Barnabé Monnot, Xyn Sun, Cairo, Mike Neuder, William X, Pranav Garimidi, and many others for insightful discussions throughout the development of this idea.

tl;dr

: I introduce a novel mechanism for enforcing proposer commitments in Ethereum without altering the existing consensus protocol.

Glossary

- PEPC-DVT

: Stands for “Protocol-Enforced Proposer Commitments - Distributed Validator Technology.” This is a framework designed to ensure that block proposers in Ethereum fulfill specific commitments without requiring changes to the existing consensus algorithm.

- Validator

: Refers to an entity, identified by a public key, that participates in Ethereum’s proof-of-stake consensus mechanism to validate transactions and create new blocks.

- Proposer

: A specialized role within the set of Validators. A Proposer is a Validator chosen to create a new block for a specific time slot in the blockchain.

- Distributed Validator (DV)

: This is a collective of individuals or nodes that share the responsibilities of a single Validator. The Validator’s private key is divided among these participants using secret-sharing techniques, ensuring that no complete signature can be generated without approval from a majority of the group. The distributed validator client is the software that enables participation in a Distributed Validator setup.

- In-Protocol Commitments

: These are a specific type of commitment that are directly related to the roles and responsibilities within the Ethereum protocol. An example is a commitment to propose a block with a certain attribute.

What problem does PEPC-DVT address?

The Problem Scenario

Consider two parties, Alice and Bob, who wish to engage in a contractual agreement. Alice promises to include Bob’s transaction in the next block she proposes on Ethereum mainnet. Furthermore, she commits to placing it as the first transaction in that block. In return, Bob agrees to pay Alice, but only if she fulfills both conditions.

The Shortcoming of Current Systems

When Alice’s turn comes to propose a block, she includes Bob’s transaction but fails to place it as the first transaction in the block. As a result, she violates her commitment and does not receive the payment from Bob. However, the Ethereum consensus protocol still validates and adds this block to the blockchain, despite the broken commitment.

The Core Issue

The existing system neither enforces nor acknowledges Alice’s commitment to Bob. Alice must be “trusted” to fulfill her end of the bargain. If she calculates that the benefits of not adhering to the commitment outweigh Bob’s payment, she has a rational incentive to cheat.

Economic Consequences

This lack of effective enforcement leads to high contracting costs and creates an environment with strategic uncertainty. Additionally, Alice has access to information that Bob doesn’t about what’s going to happen, possibly allowing her to exploit this asymmetry for her benefit.

What does PEPC-DVT do?

PEPC-DVT is designed to enforce the commitments made by block proposers in the Ethereum network. It does so by making the validity of a block dependent on whether the proposer’s commitments are met. The system utilizes Distributed Validator Technology to achieve this without altering the existing Ethereum consensus protocol.

Key Components

- Validator Key Shares

: The validator's private key is divided into parts, known as "shares" using an algorithm like Shamir's Secret Sharing. The validator retains 50% of these shares, while the remaining 50% are distributed among a network of specialized nodes called Distributed Validator Nodes.

- Commitment Specifications

: These nodes run a specialized client that on requests for their signature it checks that the data being signed satisfies the validator's commitments.

- Smart Contract in EVM

: The client interacts with a smart contract in the Ethereum Virtual Machine (EVM) to verify if the commitments are satisfied.

- Gas Limit

: To prevent abuse, a gas limit is set for the computational resources used to check commitments.

Detailed Workflow

1. Commitment Setup

: Alice and Bob agree on their respective commitments and record them in a smart contract within the EVM. Bob also escrows the payment in a contract.

1. Validator Key Distribution

: Alice divides her validator key into shares. She keeps half and distributes the other half to the Distributed Validator Nodes.

1. Block Proposal

: When it's Alice's turn to propose a block, she creates a SignedBeaconBlock and broadcasts it to the network.

1. Commitment Verification

: The Distributed Validator Nodes receive this block and initiate a verification process. Each node's client software communicates with the EVM to check if the block satisfies Alice's commitments.

1. Signature Provision

: Based on the verification result, two scenarios can occur:

- Case 1

: If the block satisfies Alice's commitments, the nodes contribute their share of the signature, enabling Alice to obtain sufficient signature shares to get the validator's signature. This allows her to achieve her goal of having the block be recognized by the protocol.

- Case 2

: If the block doesn't satisfy Alice's commitments, the nodes withhold their signature. Consequently, the validator's signature is not achieved and Ethereum consensus doesn't recognize the block.

Reliability and Risks

The system's reliability hinges on the integrity of the Distributed Validator Nodes. If a majority of these nodes are compromised, they could falsely validate a block that doesn't meet the commitments. However, this risk is mitigated by distributing the validator key shares over a decentralized network of nodes. It's also important to note that this risk doesn't find its way inside the protocol (i.e., PEPC-DVT belongs to the Diet PEPC family of solutions, which are out-of-protocol), so I don't see how a new type of risk would be introduced for the protocol.

Why PEPC-DVT?

Unique Advantages

- Seamless Integration

: No changes to Ethereum's consensus layer.

- Commitment Versatility

: Supports diverse commitment use-cases.

- Robust Security

: Uses distributed validator technology for secure commitment enforcement.

- Bounded Resource Usage

: Bounds social cost associated with evaluating some user's commitments.

- Transparency

: On-chain verification enhances accountability and facilitates credible contracting between agents.

Broader Implications

- Agent-Based Programmability at Consensus

: Facilitates programmability for in-protocol behavior, allowing for a more dynamic and responsive consensus layer.

- Economic Incentives

: Could introduce new revenue models for validators.

- Interoperability

: Potential for cross-chain transactions with other blockchain platforms.

Emily: A Protocol for Credible Commitments

Emily offers a robust and efficient way to manage commitments within the EVM. It not only simplifies the process for distributed validators but also ensures that computational resources are effectively managed. It handles the logic for determining whether some user's commitments are satisfied on behalf of the distributed validator clients, allowing them to find this out by just simulating a call to the smart contract.

Core Components

- Commitment Manager

: Central smart contract that orchestrates the commitment process.

- Commitment Structure

: Defines the properties of a commitment.

- Commitment Library

: Contains methods for evaluating and finalizing commitments.

Commitment Manager

The Commitment Manager is a smart contract that serves as the backbone of Emily. It performs two key functions:

1. Creating Commitments

: Allows any EVM address to make new commitments.

1. Evaluating Commitments

: Checks if a given value satisfies the conditions of a user's commitments.

Users can create commitments without incurring gas costs by utilizing EIP712 signatures. Multiple commitments can also be bundled and submitted simultaneously.

Commitment Structure

A commitment is characterized by two main elements:

1. Target

: Specifies the subject matter of the commitment, similar to the concept of ‘scope’ in constraint satisfaction problems.

1. Indicator Function

: A function that returns ‘1’ if the commitment is satisfied by a given value, and ‘0’ otherwise.

```
struct Commitment { uint256 timestamp; function (bytes memory) external view returns (uint256) indicatorFunction; }
```

It is with the indicator function that the commitment extensionally defines the subset of values that satisfies it.

Commitment Library: CommitmentsLib

This library contains methods for:

1. Evaluating Commitments

: Checks if a given value satisfies an array of commitments.

1. Finalizing Commitments

: Determines if a commitment is finalized.

```
library CommitmentsLib { function areCommitmentsSatisfiedByValue(Commitment[] memory commitments, bytes calldata value) public view returns (bool); function isFinalized(Commitment memory commitments) public view returns (bool finalized); }
```

Currently, commitments are only considered probably finalized by checking if some amount of time has passed since the commitment was included. This, however, is not ideal. In practice, a better option may be for the protocol to verify a proof for the commitment’s finalization.

Resource Management

Managing computational resources is a challenge due to the EVM’s gas-based operation. To prevent abuse, Emily allocates a fixed amount of gas for evaluating any user’s array of commitments. This ensures that computational resources are capped, bounding the worst-case scenario for distributed validators.

Integrating Emily into Smart Contracts

Smart contracts that wish to enforce commitments can utilize a special modifier called Screen

after inheriting from Screener.sol

. This modifier enables functions to validate whether user actions meet the commitments of their originator.

For a practical example of how this works, refer to the sample implementation for PBS in the repository under samples/PEPC.sol

, which implements PBS in terms of commitments.

Account Abstraction (ERC4337)

The repository also includes an example that integrates commitments into ERC4337 accounts. Specifically, it screens user operations to ensure they satisfy the sender’s commitments.

As part of account abstraction, ERC4337 accounts can self-declare the contract responsible for their signature aggregator. The signature aggregator, not the account, is the one that implements the logic for verifying signatures, which can be arbitrary.

In the implementation below, the sample BLS signature aggregator has been extended to enforce commitments on user operations. In practice, a screening function is used to enforce commitments.

Here’s what integrating commitments into a SignatureAggregator looks like. Notice the that the only change is the addition of the modifier Screen

.

/ * validate signature of a single userOp * This method is called after EntryPoint.simulateValidation() returns an aggregator. * First it validates the signature over the userOp. then it return data to be used when creating the handleOps: * @param userOp the userOperation received from the user. * @return sigForUserOp the value to put into the signature field of the userOp when calling handleOps. * (usually empty, unless account and aggregator support some kind of "multisig" /*

```
function validateUserOpSignature(UserOperation calldata userOp) external view Screen(userOp.sender, this.validateUserOpSignature.selector, abi.encode(userOp)) returns (bytes memory sigForUserOp) { uint256[2] memory signature = abi.decode(userOp.signature, (uint256[2])); uint256[4] memory pubkey = getUserOpPublicKey(userOp); uint256[2] memory message = _userOpToMessage(userOp, _getPublicKeyHash(pubkey));

require(BLSOpen.verifySingle(signature, pubkey, message), "BLS: wrong sig");
return "";

}
```

Token Bound Accounts (ERC6551)

The same commitment-enforcing logic has been applied to token-bound accounts, which is carried out by a slight modification in the executeCall

function. Notice the modifier.

```
/// @dev executes a low-level call against an account if the caller is authorized to make calls function executeCall(address to, uint256 value, bytes calldata data) external payable onlyAuthorized onlyUnlocked Screen(address(this), this.executeCall.selector, abi.encode(to, value, data)) returns (bytes memory) { emit TransactionExecuted(to, value, data);

_incrementNonce();

return _call(to, value, data);

}
```

This change ensures that whenever a call is executed by the account, it satisfies the account's commitments.

Challenges and Security

Challenges

1. Dependency on Distributed Validators

: The system's effectiveness is contingent on the reliability and honesty of distributed validator nodes.

1. Gas Griefing Risks

: There's a potential for malicious actors to exploit the system by consuming excessive gas, thereby affecting its performance.

1. Network Latency Concerns

: The time delay in transmitting data across the network could impact the system's efficiency and responsiveness.

Security Measures

1. Node Decentralization

: To mitigate the risk of collusion or a single point of failure, validator nodes would need to be credibly decentralized.

1. Gas Limit for Commitment Verification

: A predefined maximum amount of gas is allocated for checking commitments, preventing gas griefing attacks.

1. Local Commitment Validation

: Commitment checks are performed locally by the execution client, enhancing security and reducing latency.

Resources

- Work-in-progress specs for a PEPC distributed validator: PEPC-DVT Specs
- Protocol for credible commitments: Emily

Your feedback is highly appreciated! Feel free to reach out viatwitter.