BEOSIN
Blockchain Security

OverFoundation

Smart Contract Security Audit

No. 202405161539

May 16th, 2024

SECURING BLOCKCHAIN ECOSYSTEM

WWW.BEOSIN.COM

Contents

Summary of Audit Result

After auditing, 1 Low and 1 Info items were identified in the OverFoundation project. Specific audit details will be presented in the Findings section. Users should pay attention to the following aspects when interacting with this project:

Low	**Fixed : 1**	**Acknowledged: 0**
Info	**Fixed : 0**	**Acknowledged: 1**

Business overview

The main audit this time is the chronos part of the OverFoundation project. The Chronos is a fork of the Ethereum beacon chain Prysm, a proof-of-stake consensus client written in Go and based on the Over protocol. The total supply of Over tokens within 10 years is 1 billion, with 40% of tokens expected to be distributed in the form of consensus rewards, and may be included in reserves as needed. Compared to the 32 ETH staking requirement in Prysm, validators in Chronos need to stake 256 Over tokens to become validators.

The project introduces the concept of Deposit plans, currently divided into three stages: Early Deposit plan, Later Deposit plan, and Final Deposit plan. Validator churn limits will be adjusted based on the current Deposit plan. When the current Deposit quantity exceeds the target, the exit limit will increase by one. When the current Deposit quantity is less than the target, the pending limit will increase by one. Validator consensus rewards will also be adjusted based on the current Deposit plan. When the current Deposit quantity exceeds the target, consensus rewards decrease. When the current Deposit quantity is less than the target, consensus rewards increase. This incentivizes users to stake when staking falls below the standard.

The project introduces a reserve system. If the current consensus rewards payable to validators are less than the expected issuance of Over tokens, the excess consensus rewards will be included in the reserve. If the current consensus rewards payable to validators exceed the expected issuance of Over tokens, the shortfall will be filled from the reserve.

The project adds a bailout mechanism, which adds a BailOutScores based on the existing InactivityScores. When validators do not participate in consensus, BailOutScores accumulate, and if the accumulated score exceeds a certain threshold, validators will be ejected from the system.

1 Overview

1.1 Project Overview

Project Name	OverFoundation
Audit Scope	https://github.com/superblock-dev/chronos
Commit Hash	786039ee482718831047b98f6bc9e4e2e8016d04

1.2 Audit Overview

Audit work duration: Apr 9, 2024 – May 16, 2024

Audit team: Beosin Security Team

1.3 Audit Method

The audit methods are as follows:

1. Formal Verification

Formal verification is a technique that uses property-based approaches for testing and verification. Property specifications define a set of rules using Beosin's library of security expert rules. These rules call into the contracts under analysis and make various assertions about their behavior. The rules of the specification play a crucial role in the analysis. If the rule is violated, a concrete test case is provided to demonstrate the violation.

2. Manual Review

Using manual auditing methods, the code is read line by line to identify potential security issues. This ensures that the contract's execution logic aligns with the client's specifications and intentions, thereby safeguarding the accuracy of the contract's business logic.

The manual audit is divided into three groups to cover the entire auditing process:

The Basic Testing Group is primarily responsible for interpreting the project's code and conducting comprehensive functional testing.

The Simulated Attack Group is responsible for analyzing the audited project based on the collected historical audit vulnerability database and security incident attack models. They identify potential attack vectors and collaborate with the Basic Testing Group to conduct simulated attack tests.

The Expert Analysis Group is responsible for analyzing the overall project design, interactions with third parties, and security risks in the on-chain operational environment. They also conduct a review of the entire audit findings.

3. Static Analysis

Static analysis is a method of examining code during compilation or static analysis to detect issues. Beosin-VaaS can detect more than 100 common smart contract vulnerabilities through static analysis, such as reentrancy and block parameter dependency. It allows early and efficient discovery of problems to improve code quality and security.

2 Findings

Index	Risk description	Severity level	Status
OverFoundation-01	Logical problem of boundary update	Low	Fixed
OverFoundation-02	Potential memory leak risk	Info	Acknowledged

Finding Details:

[OverFoundation-01] Logical problem of boundary update

Severity Level	Low
Lines	Beacon-chain/core/helpers/beacon_committee.go#299-336
Description	In the *UpdateCommitteeCache* function, a single update will cache updates for both epoch and epoch+1. However, if there is a cache for the current epoch, the update for epoch+1 will be skipped and the function will return directly. This can cause problems in the update logic in *handleEpochBoundary*. For example, if it is the last slot of epoch 100, *handleEpochBoundary* will update the committecache and proposer for both 101 and 102. However, if the committecache for epoch 100 and epoch 101 has already been updated at epoch 99, *UpdateCommitteeCache* will skip this update for epoch 102 and continue executing subsequent functions. In the subsequent second *UpdateProposerIndicesInCache* function, since *UpdateCommitteeCache* did not update the committecache for epoch 102, when calling *ActiveValidatorIndices* to find the validator index, it will not be able to find the specific index, and it will traverse the current active validator to add a new index, and call *UpdateCommitteeCache* to update the committecache for epoch 102 and epoch 103. Here, the accidental addition of the committecache for epoch 103 and the failure to process the data such as the proposer for epoch 103 may cause potential problems in the future.

```
func UpdateCommitteeCache(ctx context.Context, state
state.ReadOnlyBeaconState, epoch primitives.Epoch) error {
    for _, e := range []primitives.Epoch{epoch, epoch + 1} {
        seed, err := Seed(state, e,
params.BeaconConfig().DomainBeaconAttester)
        if err != nil {
            return err
        }
        if committeeCache.HasEntry(string(seed[:])) {
            return nil
        }

        shuffledIndices, err := ShuffledIndices(state, e)
        if err != nil {
            return err
        }
        count := SlotCommitteeCount(uint64(len(shuffledIndices)))
        // Store the sorted indices as well as shuffled indices. In current spec,
        // sorted indices is required to retrieve proposer index. This is also
        // used for failing verify signature fallback.
        sortedIndices := make([]primitives.ValidatorIndex, len(shuffledIndices))
        copy(sortedIndices, shuffledIndices)
        sort.Slice(sortedIndices, func(i, j int) bool {
            return sortedIndices[i] < sortedIndices[j]
        })
        if err := committeeCache.AddCommitteeShuffledList(ctx, &cache.Committees{
            ShuffledIndices: shuffledIndices,
            CommitteeCount:  uint64(params.BeaconConfig().SlotsPerEpoch.Mul(count)),
            Seed:            seed,
            SortedIndices:   sortedIndices,
        }); err != nil {
            return err
        }
    }
    return nil
}
```

Recommendation	It is recommended to continue adding the committeeCache for the next epoch when it is detected that the committeeCache for the current epoch is not empty.
Status	**Fixed.** The project team has integrated the fix patch for version 4.0.8.

[OverFoundation-02] Potential memory leak risk

Severity Level	Info
Lines	Beacon-chain/core/helpers/beacon_committee.go#385-390

Description	In rewards_penalties.go, although balanceWithQueueCache is modified according to the original balanceCache logic, it does not release the cache in the *ClearCache* function, which may lead to a certain amount of memory accumulation. However, since the *ClearCache* function is not currently called outside of the test file, the probability of memory leakage is not high.
	`func ClearCache() {`
	`    committeeCache.Clear()`
	`    proposerIndicesCache.Clear()`
	`    syncCommitteeCache.Clear()`
	`    balanceCache.Clear()`
	`}`

Recommendation	It is recommended to add the cleanup of balanceWithQueueCache in *ClearCache*.
Status	**Acknowledged**.

3 Blockchain Audit Contents

Chronos is a beacon chain that manages validators, screens consensus committees, processes consensus rewards and penalties, and verifies blocks. It is a chain composed of beacon nodes dedicated to serving execution nodes, with each execution node corresponding to a beacon node.

3.1 JSON RPC Security Audit

3.1.1 JSON RPC Introduction

➢ **Introduction to Beacon RPC Functions**

A. Basic Concepts

RPC (Remote Procedure Call): A protocol that enables programs to call functions on a remote server over a network, as if they were local functions.

Beacon Chain: The core chain of Over protocol, responsible for coordinating the entire Chronos system, including validator management, block proposals, and consensus protocols.

B. Role of Beacon RPC

Beacon RPC is the API interface of the Beacon Chain, allowing developers to interact with it. It provides a set of APIs for reading data from the Beacon Chain and executing certain operations. These APIs are crucial for developers to build and maintain Over protocol applications.

C. Main Functions

Beacon RPC provides a wide range of functions to assist in managing the lifecycle and operations of validators. Some key functions are:

- Reading state information: Through Beacon RPC, developers can obtain various state information from the Beacon Chain, such as on-chain block information, validator status, network parameters, etc.
- Subscribing to events: Allows applications to subscribe to specific events on the Beacon Chain, such as new block proposals, validator changes, etc. This is highly useful for real-time monitoring and responding to network events.
- Sending transactions: Beacon RPC supports sending transactions to the Beacon Chain, including validator registration, exit, or attestation.
- Accessing historical data: Allows querying historical data from the Beacon Chain, including past blocks, voting records, etc.

- Validator management: Developers can manage validators through the Beacon RPC interface, including querying validator status and performance, submitting validator actions, etc.

D. Common Beacon RPC Endpoints

Here are some examples of commonly used Beacon RPC endpoints:

/beacon/state/{state_id}: Retrieves information for a specific state.

/beacon/blocks/{block_id}: Retrieves information for a specific block.

/beacon/validators: Retrieves the status information of all current validators.

/beacon/validators/{validator_id}: Retrieves information for a specific validator.

/beacon/committees/{epoch}: Retrieves committee information for a specified epoch.

/beacon/pool/attestations: Retrieves attestations from the pool.

/beacon/pool/attestations (POST): Submits attestations to the pool.

➢ Introduction to Validator RPC Functions

A. Basic Concepts

RPC (Remote Procedure Call): A method to call remote services over a network, enabling developers to execute program functions remotely.

Validator: In Chronos, validators are entities responsible for proposing and validating blocks. Validators participate in network consensus by staking Over tokens and receive rewards.

B. Role of Validator RPC

Validator RPC is a set of API interfaces provided by the Beacon Chain specifically for managing and performing operations related to validators. These interfaces allow developers and validator nodes to interact with the Beacon Chain to execute various validator functions.

C. Main Functions

Validator RPC provides a wide range of functions to assist in managing the lifecycle and operations of validators. Some key functions are:

- Validator Registration and Activation: Through the RPC interface, validators can register and activate their identity to start participating in network consensus.
- Validator Status Query: Provides an interface to query validator status, including its current state (active, exiting, pending activation, etc.) and performance.
- Submitting Attestations: Validators need to submit attestations to confirm the validity of blocks, and these operations can be completed through the RPC interface.

- Proposing Blocks: When selected, validators need to propose new blocks, which is also done through the RPC interface.
- Managing Rewards and Penalties: Validators can query their reward and penalty status through the RPC interface to understand their earning and penalty records

D. Common Validator RPC Endpoints

Here are some examples of commonly used Validator RPC endpoints:

/eth/v1/validator/duties/attester/{epoch}: Retrieves the attester duties for a specific epoch.

/eth/v1/validator/duties/proposer/{epoch}: Retrieves the proposer duties for a specific epoch.

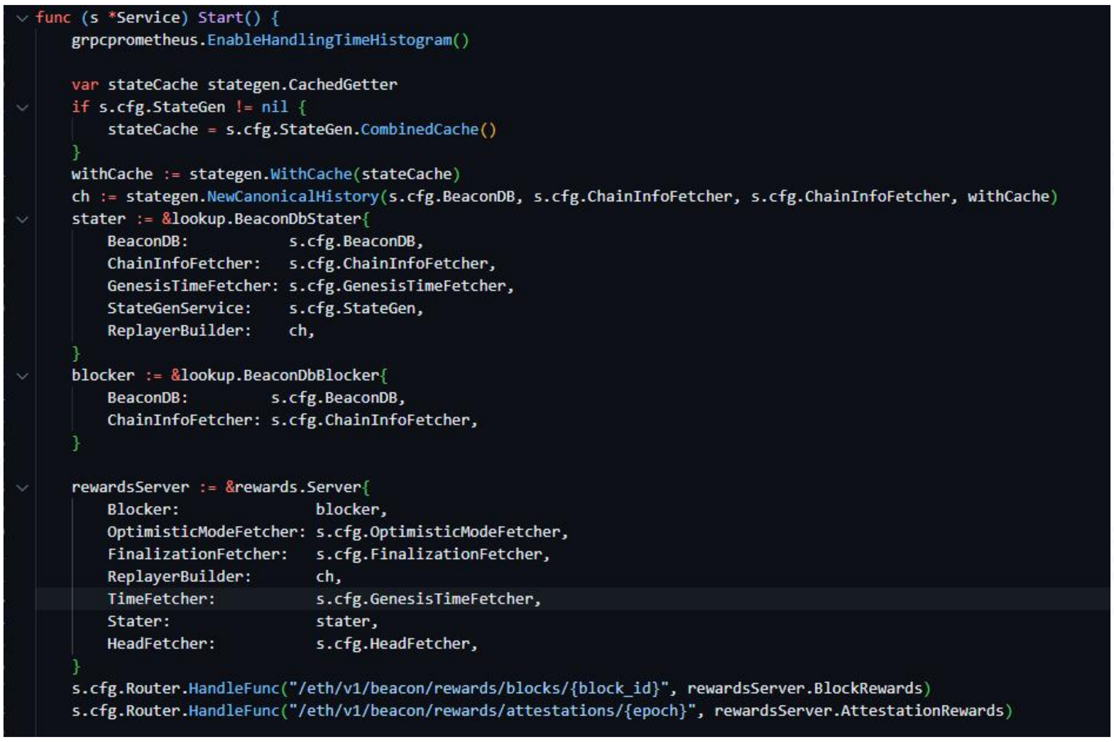
/eth/v1/validator/attestations (POST): Submits an attestation from a validator.

/eth/v1/validator/blocks (POST): Submits a block proposal from a validator.

/eth/v1/validator/aggregate_and_proofs (POST): Submits aggregated attestations and related data.

3.1.2 JSON RPC Processing Logic

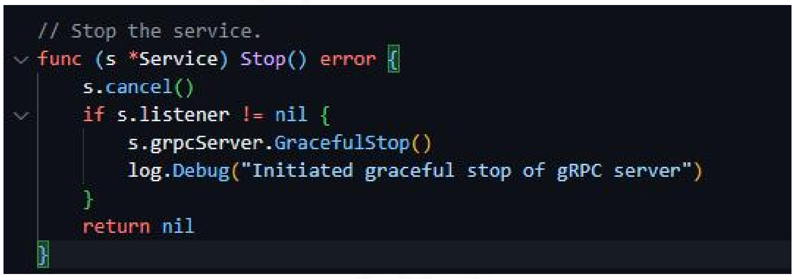
The Beacon RPC code is located under the beacon-chain/rpc path, and the startup logic for the RPC is implemented in the Start method of the service.go file. This Start method initializes and launches multiple services related to the beacon chain and validators, including state management, blocks and validator rewards, node information, log streaming, and more. It sets up the corresponding HTTP and GRPC endpoints, registers all the services to the GRPC server, enables debugging and additional node RPC endpoints based on the configuration, and finally starts the GRPC server to handle external requests. The specific code is shown in the following diagram:



```go
func (s *Service) Start() {
    grpcprometheus.EnableHandlingTimeHistogram()

    var stateCache stategen.CachedGetter
    if s.cfg.StateGen != nil {
        stateCache = s.cfg.StateGen.CombinedCache()
    }
    withCache := stategen.WithCache(stateCache)
    ch := stategen.NewCanonicalHistory(s.cfg.BeaconDB, s.cfg.ChainInfoFetcher, s.cfg.ChainInfoFetcher, withCache)
    stater := &lookup.BeaconDbStater{
        BeaconDB:           s.cfg.BeaconDB,
        ChainInfoFetcher:   s.cfg.ChainInfoFetcher,
        GenesisTimeFetcher: s.cfg.GenesisTimeFetcher,
        StateGenService:    s.cfg.StateGen,
        ReplayerBuilder:    ch,
    }
    blocker := &lookup.BeaconDbBlocker{
        BeaconDB:         s.cfg.BeaconDB,
        ChainInfoFetcher: s.cfg.ChainInfoFetcher,
    }

    rewardsServer := &rewards.Server{
        Blocker:               blocker,
        OptimisticModeFetcher: s.cfg.OptimisticModeFetcher,
        FinalizationFetcher:   s.cfg.FinalizationFetcher,
        ReplayerBuilder:       ch,
        TimeFetcher:           s.cfg.GenesisTimeFetcher,
        Stater:                stater,
        HeadFetcher:           s.cfg.HeadFetcher,
    }
    s.cfg.Router.HandleFunc("/eth/v1/beacon/rewards/blocks/{block_id}", rewardsServer.BlockRewards)
    s.cfg.Router.HandleFunc("/eth/v1/beacon/rewards/attestations/{epoch}", rewardsServer.AttestationRewards)
```

Figure 1 Screenshot of *Start* function

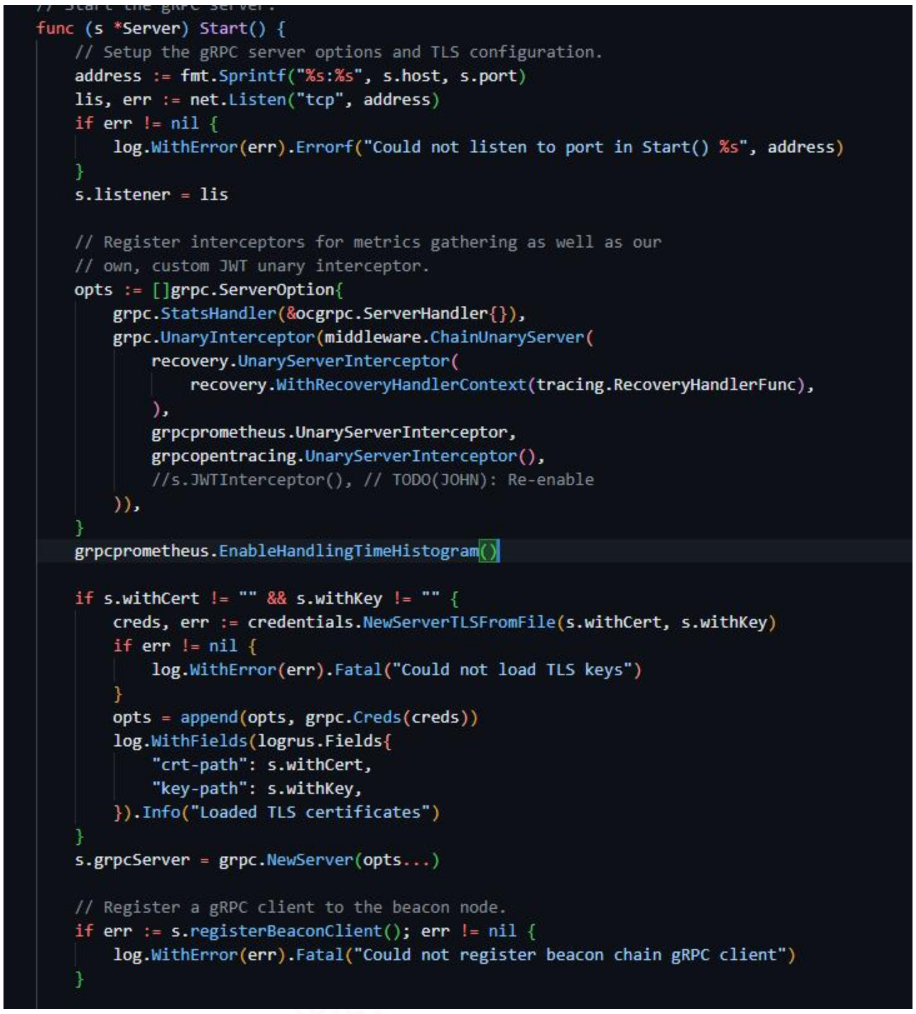
The Stop method is used to stop the service. It first calls s.cancel() to cancel the context, terminating any ongoing operations. Then, if s.listener exists, it calls s.grpcServer.GracefulStop() to stop the gRPC server gracefully, and logs a debug message indicating that the gRPC server has started shutting down. Finally, the method returns nil, indicating that the operation has successfully completed.



```go
// Stop the service.
func (s *Service) Stop() error {
    s.cancel()
    if s.listener != nil {
        s.grpcServer.GracefulStop()
        log.Debug("Initiated graceful stop of gRPC server")
    }
    return nil
}
```

Figure 2 Screenshot of *Stop* function

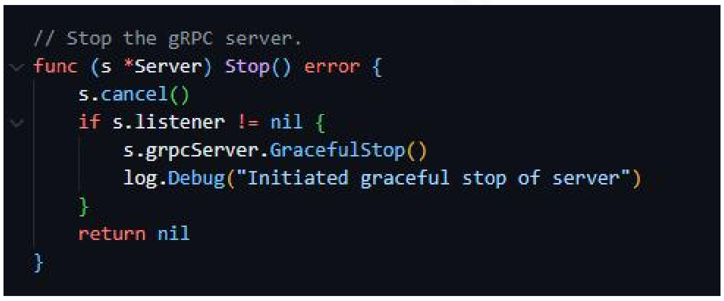
The Validator RPC code is located under the validator\rpc\ path, and the startup logic for the RPC is implemented in the Start method of the service.go file. This function launches a gRPC server. First, it sets up the gRPC server's options and TLS configuration, listening on a specified address and port. Then, it registers various interceptors, including interceptors for metrics and error recovery, and loads the TLS configuration from provided certificates. After that, the function registers a client to the beacon node and multiple gRPC services, such as authentication, wallet, health checking, and so on. If the isOverNode flag is true, it will also register the OverNode service. Finally, it starts the gRPC server to begin listening for client requests.



```go
func (s *Server) Start() {
    // Setup the gRPC server options and TLS configuration.
    address := fmt.Sprintf("%s:%s", s.host, s.port)
    lis, err := net.Listen("tcp", address)
    if err != nil {
        log.WithError(err).Errorf("Could not listen to port in Start() %s", address)
    }
    s.listener = lis

    // Register interceptors for metrics gathering as well as our
    // own, custom JWT unary interceptor.
    opts := []grpc.ServerOption{
        grpc.StatsHandler(&ocgrpc.ServerHandler{}),
        grpc.UnaryInterceptor(middleware.ChainUnaryServer(
            recovery.UnaryServerInterceptor(
                recovery.WithRecoveryHandlerContext(tracing.RecoveryHandlerFunc),
            ),
            grpcprometheus.UnaryServerInterceptor,
            grpcopentracing.UnaryServerInterceptor(),
            //s.JWTInterceptor(), // TODO(JOHN): Re-enable
        )),
    }
    grpcprometheus.EnableHandlingTimeHistogram()

    if s.withCert != "" && s.withKey != "" {
        creds, err := credentials.NewServerTLSFromFile(s.withCert, s.withKey)
        if err != nil {
            log.WithError(err).Fatal("Could not load TLS keys")
        }
        opts = append(opts, grpc.Creds(creds))
        log.WithFields(logrus.Fields{
            "crt-path": s.withCert,
            "key-path": s.withKey,
        }).Info("Loaded TLS certificates")
    }
    s.grpcServer = grpc.NewServer(opts...)

    // Register a gRPC client to the beacon node.
    if err := s.registerBeaconClient(); err != nil {
        log.WithError(err).Fatal("Could not register beacon chain gRPC client")
    }
```

Figure 3 Screenshot of *Start* function

This function is used to stop the server from running. First, it calls the cancel() function to cancel related functionalities, and then, if a listener exists, it shuts down the gRPC server and logs the server stoppage.

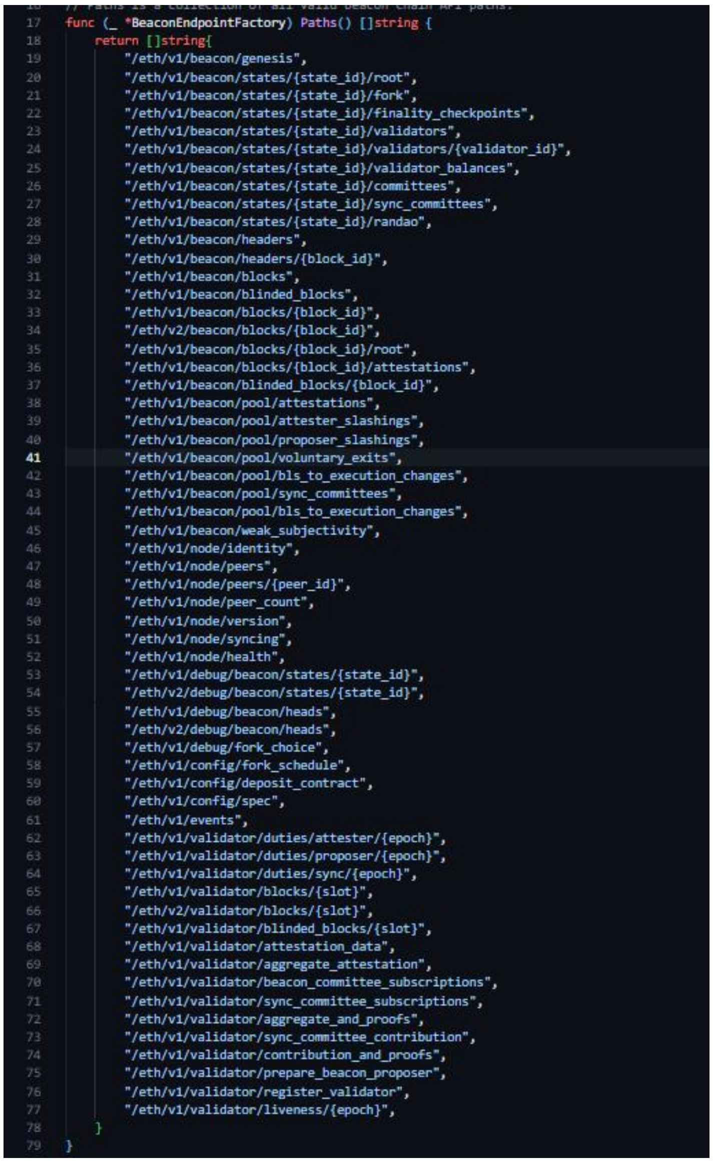


```go
// Stop the gRPC server.
func (s *Server) Stop() error {
    s.cancel()
    if s.listener != nil {
        s.grpcServer.GracefulStop()
        log.Debug("Initiated graceful stop of server")
    }
    return nil
}
```

Figure 4 Screenshot of *Stop* function

3.1.3 RPC Sensitive Interface Permission

The vast majority of the RPC interfaces currently provided by Chronos node are for querying data, and there are no high authority interfaces.

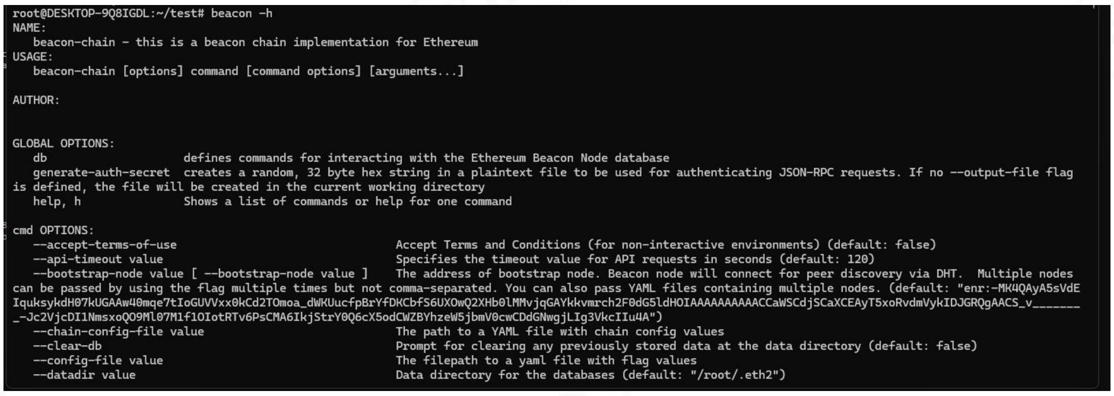


```go
func (_ *BeaconEndpointFactory) Paths() []string {
	return []string{
		"/eth/v1/beacon/genesis",
		"/eth/v1/beacon/states/{state_id}/root",
		"/eth/v1/beacon/states/{state_id}/fork",
		"/eth/v1/beacon/states/{state_id}/finality_checkpoints",
		"/eth/v1/beacon/states/{state_id}/validators",
		"/eth/v1/beacon/states/{state_id}/validators/{validator_id}",
		"/eth/v1/beacon/states/{state_id}/validator_balances",
		"/eth/v1/beacon/states/{state_id}/committees",
		"/eth/v1/beacon/states/{state_id}/sync_committees",
		"/eth/v1/beacon/states/{state_id}/randao",
		"/eth/v1/beacon/headers",
		"/eth/v1/beacon/headers/{block_id}",
		"/eth/v1/beacon/blocks",
		"/eth/v1/beacon/blinded_blocks",
		"/eth/v1/beacon/blocks/{block_id}",
		"/eth/v2/beacon/blocks/{block_id}",
		"/eth/v1/beacon/blocks/{block_id}/root",
		"/eth/v1/beacon/blocks/{block_id}/attestations",
		"/eth/v1/beacon/blinded_blocks/{block_id}",
		"/eth/v1/beacon/pool/attestations",
		"/eth/v1/beacon/pool/attester_slashings",
		"/eth/v1/beacon/pool/proposer_slashings",
		"/eth/v1/beacon/pool/voluntary_exits",
		"/eth/v1/beacon/pool/bls_to_execution_changes",
		"/eth/v1/beacon/pool/sync_committees",
		"/eth/v1/beacon/pool/bls_to_execution_changes",
		"/eth/v1/beacon/weak_subjectivity",
		"/eth/v1/node/identity",
		"/eth/v1/node/peers",
		"/eth/v1/node/peers/{peer_id}",
		"/eth/v1/node/peer_count",
		"/eth/v1/node/version",
		"/eth/v1/node/syncing",
		"/eth/v1/node/health",
		"/eth/v1/debug/beacon/states/{state_id}",
		"/eth/v2/debug/beacon/states/{state_id}",
		"/eth/v1/debug/beacon/heads",
		"/eth/v2/debug/beacon/heads",
		"/eth/v1/debug/fork_choice",
		"/eth/v1/config/fork_schedule",
		"/eth/v1/config/deposit_contract",
		"/eth/v1/config/spec",
		"/eth/v1/events",
		"/eth/v1/validator/duties/attester/{epoch}",
		"/eth/v1/validator/duties/proposer/{epoch}",
		"/eth/v1/validator/duties/sync/{epoch}",
		"/eth/v1/validator/blocks/{slot}",
		"/eth/v2/validator/blocks/{slot}",
		"/eth/v1/validator/blinded_blocks/{slot}",
		"/eth/v1/validator/attestation_data",
		"/eth/v1/validator/aggregate_attestation",
		"/eth/v1/validator/beacon_committee_subscriptions",
		"/eth/v1/validator/sync_committee_subscriptions",
		"/eth/v1/validator/aggregate_and_proofs",
		"/eth/v1/validator/sync_committee_contribution",
		"/eth/v1/validator/contribution_and_proofs",
		"/eth/v1/validator/prepare_beacon_proposer",
		"/eth/v1/validator/register_validator",
		"/eth/v1/validator/liveness/{epoch}",
	}
}
```

Figure 5 Screenshots of relevant interfaces

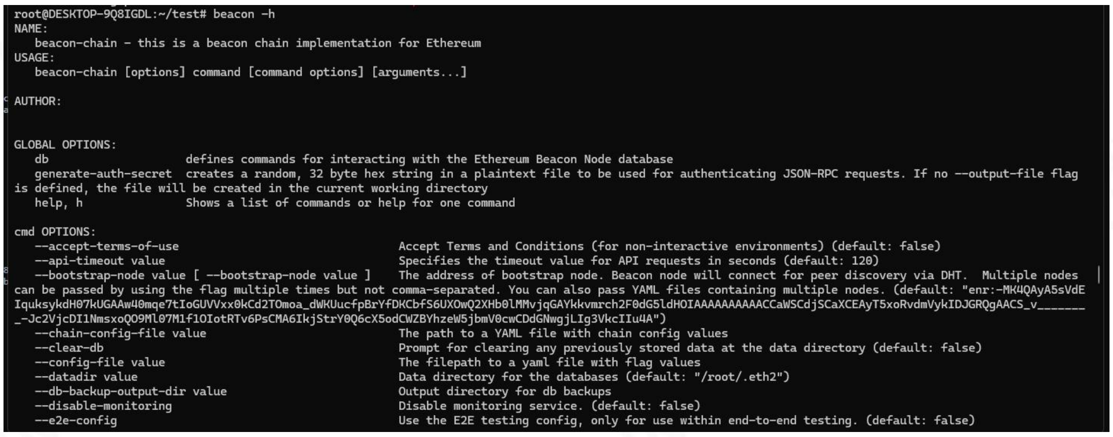
3.1.4 CLI Commands Security Audit

Below is a diagram of the execution of commands related to Beacon and validator clients:



```
root@DESKTOP-9Q8IGDL:~/test# beacon -h
NAME:
   beacon-chain - this is a beacon chain implementation for Ethereum
USAGE:
   beacon-chain [options] command [command options] [arguments...]

AUTHOR:


GLOBAL OPTIONS:
   db                    defines commands for interacting with the Ethereum Beacon Node database
   generate-auth-secret  creates a random, 32 byte hex string in a plaintext file to be used for authenticating JSON-RPC requests. If no --output-file flag
is defined, the file will be created in the current working directory
   help, h               Shows a list of commands or help for one command

cmd OPTIONS:
   --accept-terms-of-use                              Accept Terms and Conditions (for non-interactive environments) (default: false)
   --api-timeout value                                Specifies the timeout value for API requests in seconds (default: 120)
   --bootstrap-node value [ --bootstrap-node value ]    The address of bootstrap node. Beacon node will connect for peer discovery via DHT.  Multiple nodes
can be passed by using the flag multiple times but not comma-separated. You can also pass YAML files containing multiple nodes. (default: "enr:-MK4QAyA5sVdE
IquksykdH07kUGAAw40mqe7tIoGUVVxx0kCd2TOmoa_dWKUucfpBrYfDKCbfS6UXOwQ2XHb0lMMvjqGAYkkvmrch2F0dG5ldHOIAAAAAAAAACCaWSCdjSCaXCEAyT5xoRvdmVykIDJGRQgAACS_v_______
_-Jc2VjcDI1NmsxoQO9Ml07M1f1OIotRTv6PsCMA6IkjStrY0Q6cX5odCWZBYhzeW5jbmV0cwCDdGNwgjLIg3VkcIIu4A")
   --chain-config-file value                          The path to a YAML file with chain config values
   --clear-db                                         Prompt for clearing any previously stored data at the data directory (default: false)
   --config-file value                                The filepath to a yaml file with flag values
   --datadir value                                    Data directory for the databases (default: "/root/.eth2")
```

Figure 6 Screenshot of beacon client



```
root@DESKTOP-9Q8IGDL:~/test# beacon -h
NAME:
   beacon-chain - this is a beacon chain implementation for Ethereum
USAGE:
   beacon-chain [options] command [command options] [arguments...]

AUTHOR:


GLOBAL OPTIONS:
   db                     defines commands for interacting with the Ethereum Beacon Node database
   generate-auth-secret  creates a random, 32 byte hex string in a plaintext file to be used for authenticating JSON-RPC requests. If no --output-file flag
is defined, the file will be created in the current working directory
   help, h                Shows a list of commands or help for one command

cmd OPTIONS:
   --accept-terms-of-use                              Accept Terms and Conditions (for non-interactive environments) (default: false)
   --api-timeout value                                Specifies the timeout value for API requests in seconds (default: 120)
   --bootstrap-node value [ --bootstrap-node value ]    The address of bootstrap node. Beacon node will connect for peer discovery via DHT.  Multiple nodes
can be passed by using the flag multiple times but not comma-separated. You can also pass YAML files containing multiple nodes. (default: "enr:-MK4QAyA5sVdE
IquksykdH07kUGAAw40mqe7tIoGUVVxx0kCd2TOmoa_dWKUucfpBrYfDKCbfS6UXOwQ2XHb0lMMvjqGAYkkvmrch2F0dG5ldHOIAAAAAAAAACCaWSCdjSCaXCEAyT5xoRvdmVykIDJGRQgAACS_v_______
_-Jc2VjcDI1NmsxoQO9Ml07M1f1OIotRTv6PsCMA6IkjStrY0Q6cX5odCWZBYhzeW5jbmV0cwCDdGNwgjLIg3VkcIIu4A")
   --chain-config-file value                          The path to a YAML file with chain config values
   --clear-db                                         Prompt for clearing any previously stored data at the data directory (default: false)
   --config-file value                                The filepath to a yaml file with flag values
   --datadir value                                    Data directory for the databases (default: "/root/.eth2")
   --db-backup-output-dir value                       Output directory for db backups
   --disable-monitoring                               Disable monitoring service. (default: false)
   --e2e-config                                       Use the E2E testing config, only for use within end-to-end testing. (default: false)
```

Figure 7 Screenshot of validator client

After testing, the relevant commands in the CLI meet the audit requirements and there is no security risk.

3.2 Node Security

3.2.1 basic concept

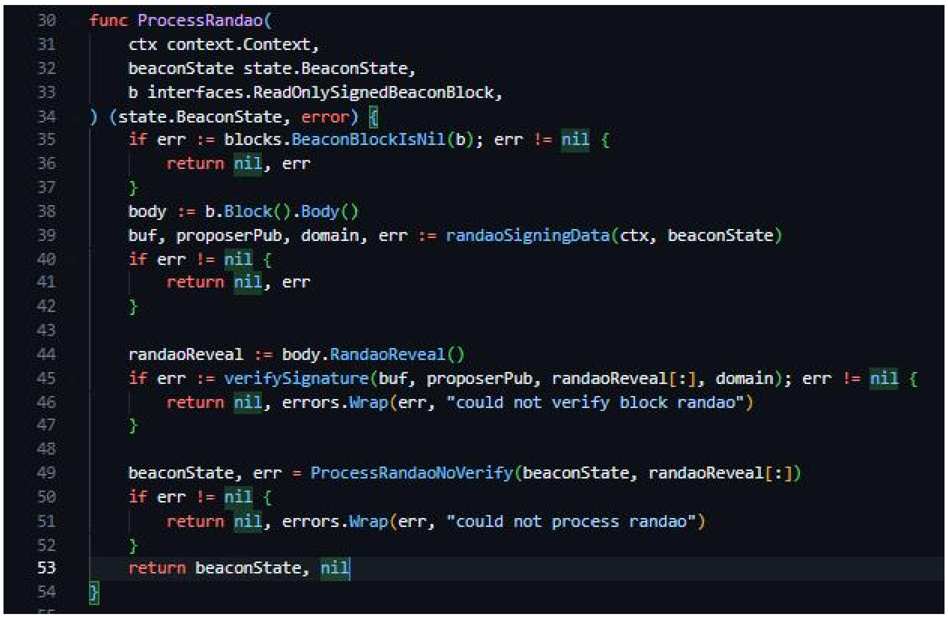
Slot: A unit of time in Chronos, with one slot occurring every 12 seconds. In each slot, a block can be proposed.

Epoch: A group of consecutive slots, typically comprising 32 slots, forming one epoch. Each epoch is approximately 6.4 minutes.

3.2.2 Diversification of risk

Random Selection of Validators: Each slot randomly selects a validator to propose a block. This random selection process is based on RANDAO and VDF randomness generation mechanisms, ensuring that the selection of validators is unpredictable and reducing the possibility of attackers knowing the proposer ahead of time.

Uniform Distribution of Responsibilities: Validators assume different roles (proposer or participant) in different slots and epochs, evenly distributing responsibilities and preventing a single validator from occupying an important position for a long time.



```go
func ProcessRandao(
	ctx context.Context,
	beaconState state.BeaconState,
	b interfaces.ReadOnlySignedBeaconBlock,
) (state.BeaconState, error) {
	if err := blocks.BeaconBlockIsNil(b); err != nil {
		return nil, err
	}
	body := b.Block().Body()
	buf, proposerPub, domain, err := randaoSigningData(ctx, beaconState)
	if err != nil {
		return nil, err
	}

	randaoReveal := body.RandaoReveal()
	if err := verifySignature(buf, proposerPub, randaoReveal[:], domain); err != nil {
		return nil, errors.Wrap(err, "could not verify block randao")
	}

	beaconState, err = ProcessRandaoNoVerify(beaconState, randaoReveal[:])
	if err != nil {
		return nil, errors.Wrap(err, "could not process randao")
	}
	return beaconState, nil
}
```

Figure 8 Source code for the *RANDAO* function

3.2.3 Synchronized checkpoint and reorganization protection

Checkpoint: The first block of each epoch is referred to as a checkpoint, and the network votes and confirms on the checkpoint. The checkpoint mechanism enhances the stability and security of the blockchain through an explicit voting and confirmation process, preventing long-term chain reorganizations.

Rapid Confirmation and Reorganization Limit: By using epochs as synchronization points, blocks can be confirmed more quickly, and the depth and frequency of chain reorganizations are limited, thus improving the overall network's security and certainty.

3.2.4 Refinement of incentive and penalty mechanisms

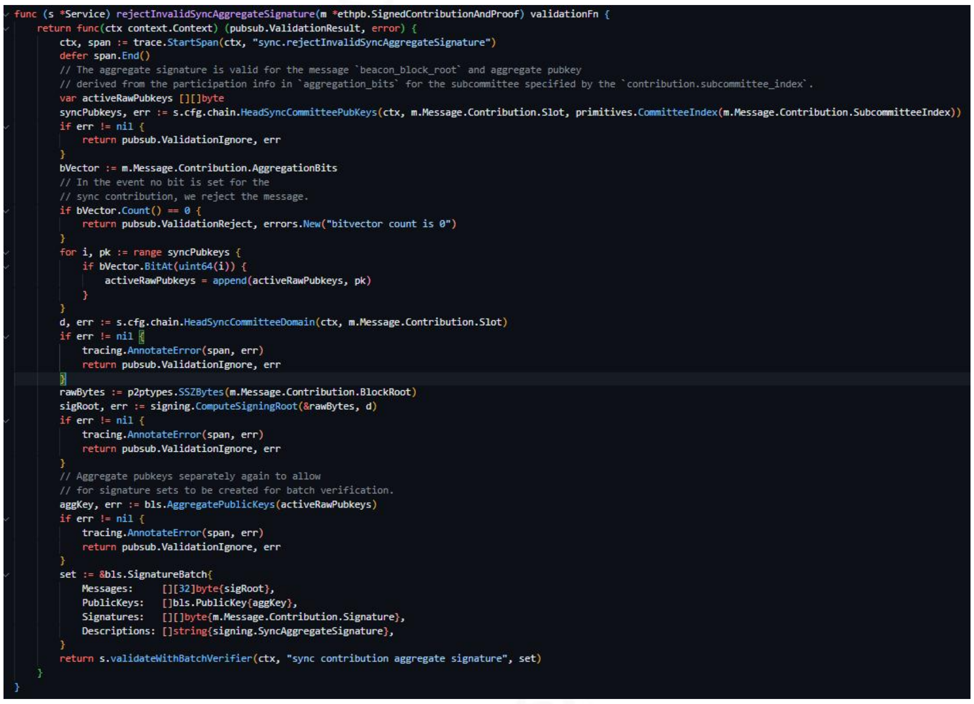
Periodic Evaluation: At the end of each epoch, the network evaluates the validators' performance, including attendance rates, proposing valid blocks, etc., and provides rewards or penalties accordingly. This periodic evaluation mechanism ensures that validators receive economic incentives or penalties within a relatively short period of time, motivating them to remain honest and actively participate.

Mitigating the Impact of Prolonged Offline Status: If validators perform poorly in an epoch, their rewards will be reduced or they may even face penalties. This mechanism mitigates the negative impact of prolonged offline status on the network.

3.2.5 Anti-Forking Attacks

Multiple Validation Mechanism: At the end of each epoch, the validity of blocks is further verified through aggregate signatures and consensus among multiple validators. This multiple validation mechanism improves resilience against fork attacks.

Rotating Validator Sets: The set of validators rotates between different epochs, and this dynamic change makes it difficult for attackers to influence a sufficient number of validators in a short period of time, thereby reducing the risk of fork attacks.



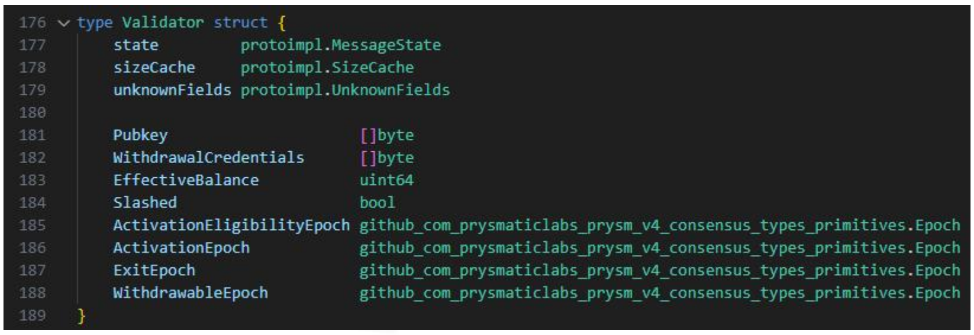
```go
func (s *Service) rejectInvalidSyncAggregateSignature(m *ethpb.SignedContributionAndProof) validationFn {
	return func(ctx context.Context) (pubsub.ValidationResult, error) {
		ctx, span := trace.StartSpan(ctx, "sync.rejectInvalidSyncAggregateSignature")
		defer span.End()
		// The aggregate signature is valid for the message `beacon_block_root` and aggregate pubkey
		// derived from the participation info in `aggregation_bits` for the subcommittee specified by the `contribution.subcommittee_index`.
		var activeRawPubkeys [][]byte
		syncPubkeys, err := s.cfg.chain.HeadSyncCommitteePubKeys(ctx, m.Message.Contribution.Slot, primitives.CommitteeIndex(m.Message.Contribution.SubcommitteeIndex))
		if err != nil {
			return pubsub.ValidationIgnore, err
		}
		bVector := m.Message.Contribution.AggregationBits
		// In the event no bit is set for the
		// sync contribution, we reject the message.
		if bVector.Count() == 0 {
			return pubsub.ValidationReject, errors.New("bitvector count is 0")
		}
		for i, pk := range syncPubkeys {
			if bVector.BitAt(uint64(i)) {
				activeRawPubkeys = append(activeRawPubkeys, pk)
			}
		}
		d, err := s.cfg.chain.HeadSyncCommitteeDomain(ctx, m.Message.Contribution.Slot)
		if err != nil {
			tracing.AnnotateError(span, err)
			return pubsub.ValidationIgnore, err
		}
		rawBytes := p2ptypes.SSZBytes(m.Message.Contribution.BlockRoot)
		sigRoot, err := signing.ComputeSigningRoot(&rawBytes, d)
		if err != nil {
			tracing.AnnotateError(span, err)
			return pubsub.ValidationIgnore, err
		}
		// Aggregate pubkeys separately again to allow
		// for signature sets to be created for batch verification.
		aggKey, err := bls.AggregatePublicKeys(activeRawPubkeys)
		if err != nil {
			tracing.AnnotateError(span, err)
			return pubsub.ValidationIgnore, err
		}
		set := &bls.SignatureBatch{
			Messages:     [][32]byte{sigRoot},
			PublicKeys:   []bls.PublicKey{aggKey},
			Signatures:   [][]byte{m.Message.Contribution.Signature},
			Descriptions: []string{signing.SyncAggregateSignature},
		}
		return s.validateWithBatchVerifier(ctx, "sync contribution aggregate signature", set)
	}
}
```

Figure 9 Source code for the *rejectInvalidSyncAggregateSignature* function

3.3 Validator & Asset Security

3.3.1 Validator Model

The Chronos uses a same validator model with Prysm, with the following basic data structure:



```go
type Validator struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Pubkey                     []byte
    WithdrawalCredentials      []byte
    EffectiveBalance           uint64
    Slashed                    bool
    ActivationEligibilityEpoch github_com_prysmaticlabs_prysm_v4_consensus_types_primitives.Epoch
    ActivationEpoch            github_com_prysmaticlabs_prysm_v4_consensus_types_primitives.Epoch
    ExitEpoch                  github_com_prysmaticlabs_prysm_v4_consensus_types_primitives.Epoch
    WithdrawableEpoch          github_com_prysmaticlabs_prysm_v4_consensus_types_primitives.Epoch
}
```

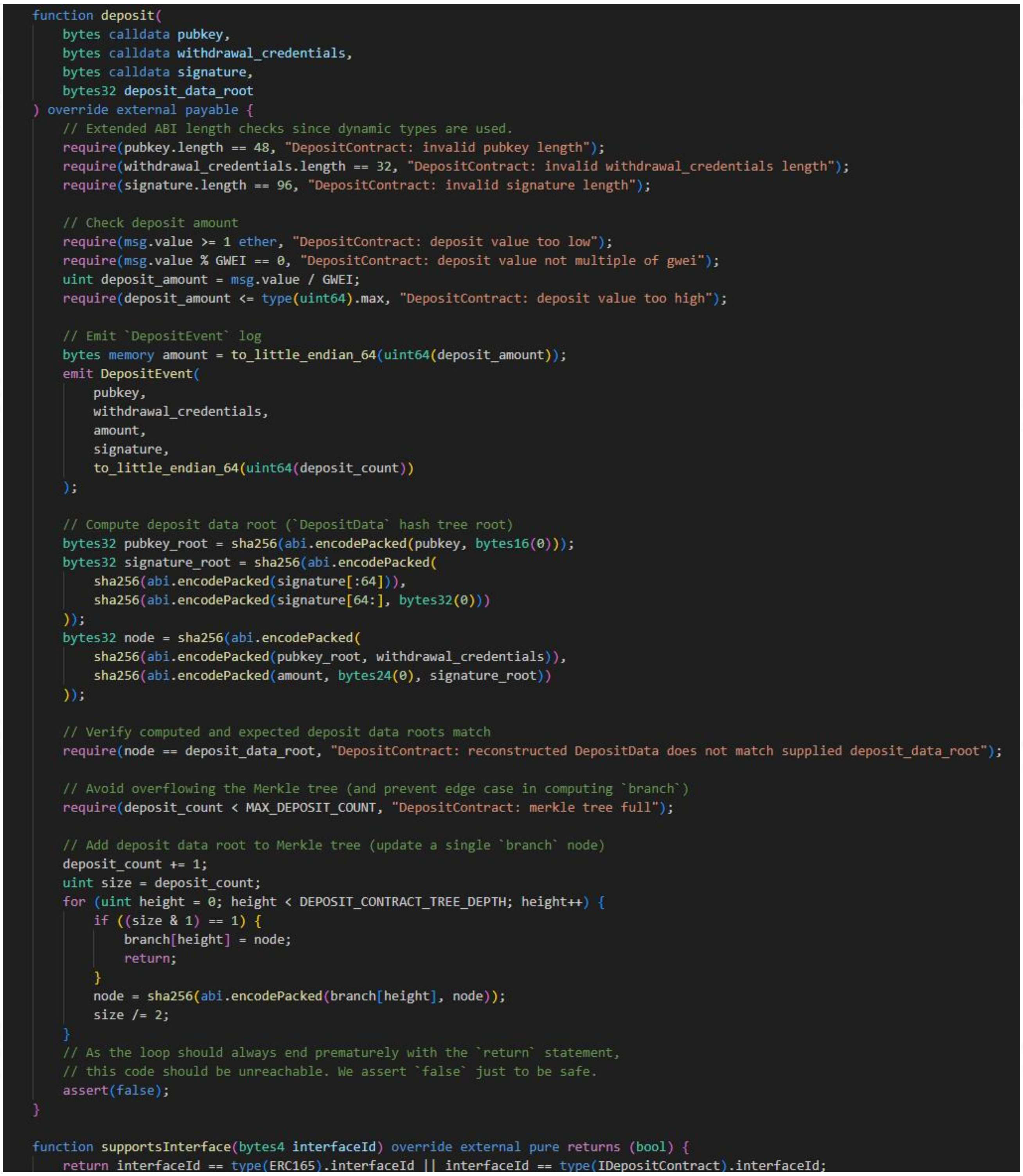
Figure 10 Source code of the data structure of Validator

- Pubkey: Used for verifier authentication;
- WithdrawalCredentials: Withdrawal credentials, used for verifying the identity of the verifier when withdrawing;
- EffectiveBalance: The current effective deposit balance;
- Slashed: Whether it is punished;
- ActivationEligibilityEpoch: Activation eligibility epoch, only after the verifier reaches this epoch can they be eligible for activation;
- ActivationEpoch: activation epoch, after which the validator is officially activated and begins to participate in consensus;
- ExitEpoch: The epoch for exiting, after which the validator can start the exit process;
- WithdrawableEpoch: withdrawable epoch, after which the validator can withdraw the balance of their collateral;

3.3.2 Validator Deposit

(1) Deposit contract

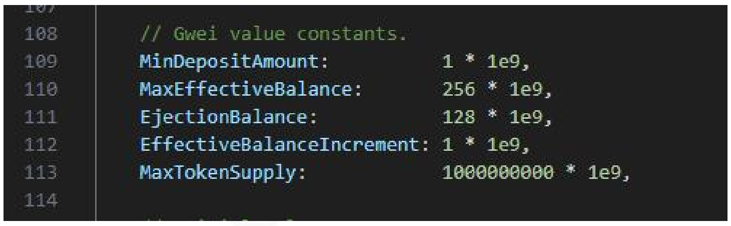
If users want to stake over code to become a validator, they need to deposit 256 Over tokens in the mainnet's deposit contract, which triggers the ProcessDeposit of the beacon chain.



```
function deposit(
    bytes calldata pubkey,
    bytes calldata withdrawal_credentials,
    bytes calldata signature,
    bytes32 deposit_data_root
) override external payable {
    // Extended ABI length checks since dynamic types are used.
    require(pubkey.length == 48, "DepositContract: invalid pubkey length");
    require(withdrawal_credentials.length == 32, "DepositContract: invalid withdrawal_credentials length");
    require(signature.length == 96, "DepositContract: invalid signature length");

    // Check deposit amount
    require(msg.value >= 1 ether, "DepositContract: deposit value too low");
    require(msg.value % GWEI == 0, "DepositContract: deposit value not multiple of gwei");
    uint deposit_amount = msg.value / GWEI;
    require(deposit_amount <= type(uint64).max, "DepositContract: deposit value too high");

    // Emit `DepositEvent` log
    bytes memory amount = to_little_endian_64(uint64(deposit_amount));
    emit DepositEvent(
        pubkey,
        withdrawal_credentials,
        amount,
        signature,
        to_little_endian_64(uint64(deposit_count))
    );

    // Compute deposit data root (`DepositData` hash tree root)
    bytes32 pubkey_root = sha256(abi.encodePacked(pubkey, bytes16(0)));
    bytes32 signature_root = sha256(abi.encodePacked(
        sha256(abi.encodePacked(signature[:64])),
        sha256(abi.encodePacked(signature[64:], bytes32(0)))
    ));
    bytes32 node = sha256(abi.encodePacked(
        sha256(abi.encodePacked(pubkey_root, withdrawal_credentials)),
        sha256(abi.encodePacked(amount, bytes24(0), signature_root))
    ));

    // Verify computed and expected deposit data roots match
    require(node == deposit_data_root, "DepositContract: reconstructed DepositData does not match supplied deposit_data_root");

    // Avoid overflowing the Merkle tree (and prevent edge case in computing `branch`)
    require(deposit_count < MAX_DEPOSIT_COUNT, "DepositContract: merkle tree full");

    // Add deposit data root to Merkle tree (update a single `branch` node)
    deposit_count += 1;
    uint size = deposit_count;
    for (uint height = 0; height < DEPOSIT_CONTRACT_TREE_DEPTH; height++) {
        if ((size & 1) == 1) {
            branch[height] = node;
            return;
        }
        node = sha256(abi.encodePacked(branch[height], node));
        size /= 2;
    }
    // As the loop should always end prematurely with the `return` statement,
    // this code should be unreachable. We assert `false` just to be safe.
    assert(false);
}

function supportsInterface(bytes4 interfaceId) override external pure returns (bool) {
    return interfaceId == type(ERC165).interfaceId || interfaceId == type(IDepositContract).interfaceId;
```

Figure 11 Source code of the *deposit* function

Please note that the requirements for the number of deposit(MaxEffectiveBalance) here are different from those for Ethereum, which is 32*1e9, and the Over protocol is 256*1e9.

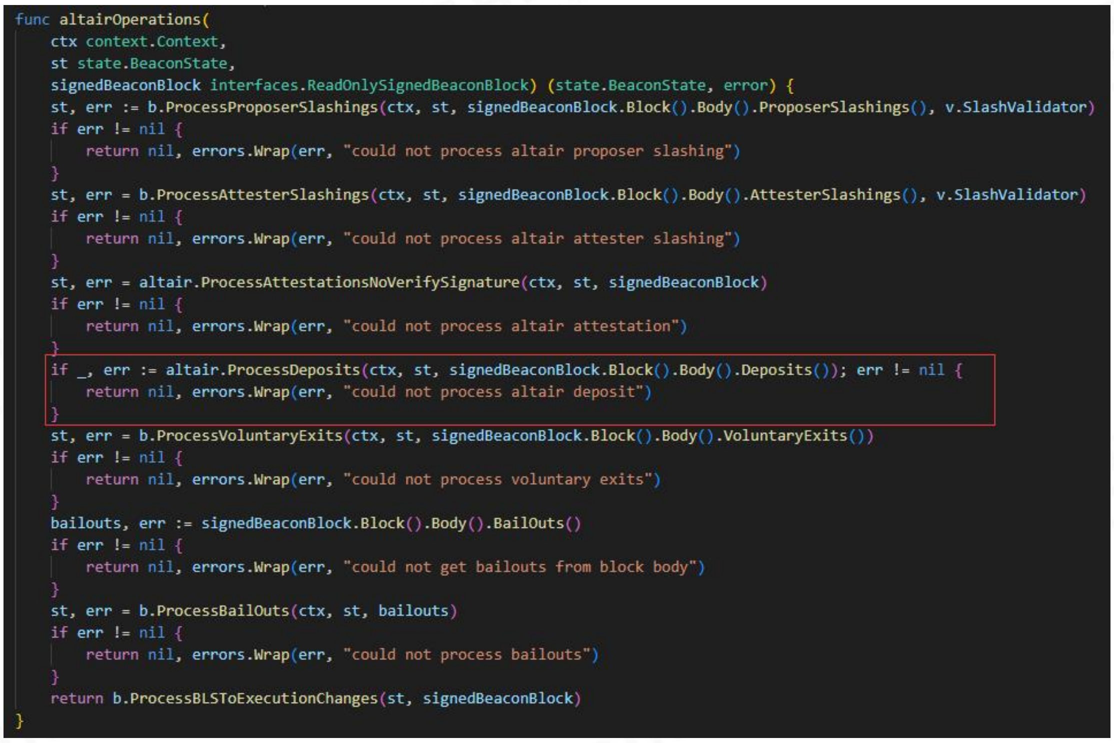


```
// Gwei value constants.
MinDepositAmount:          1 * 1e9,
MaxEffectiveBalance:       256 * 1e9,
EjectionBalance:           128 * 1e9,
EffectiveBalanceIncrement: 1 * 1e9,
MaxTokenSupply:            1000000000 * 1e9,
```

Figure 12 Config data

(2) Deposit process

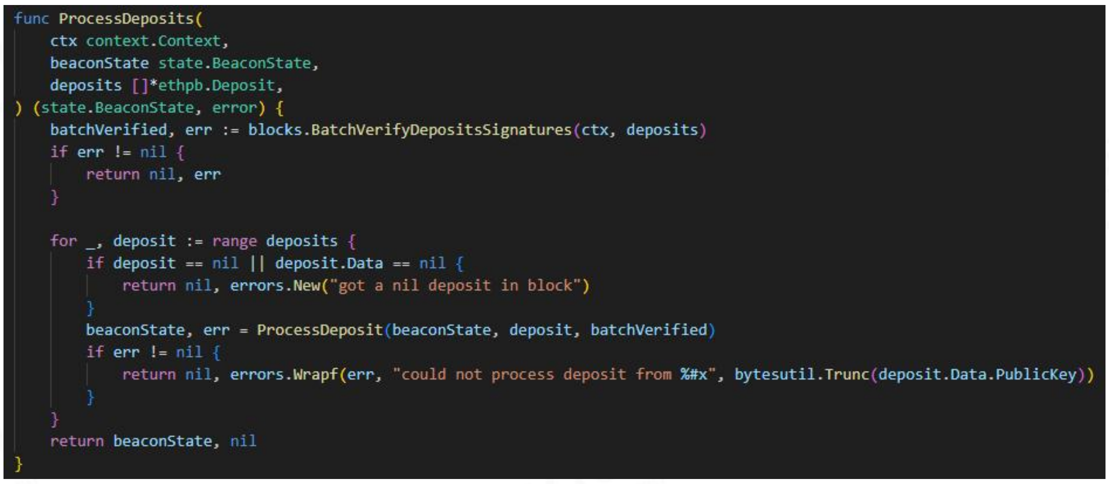
The Beacon Chain extracts the corresponding deposit data from the currently signed BeaconBlock, which is then processed in the subsequent *ProcessDeposit* function.



```
func altairOperations(
    ctx context.Context,
    st state.BeaconState,
    signedBeaconBlock interfaces.ReadOnlySignedBeaconBlock) (state.BeaconState, error) {
    st, err := b.ProcessProposerSlashings(ctx, st, signedBeaconBlock.Block().Body().ProposerSlashings(), v.SlashValidator)
    if err != nil {
        return nil, errors.Wrap(err, "could not process altair proposer slashing")
    }
    st, err = b.ProcessAttesterSlashings(ctx, st, signedBeaconBlock.Block().Body().AttesterSlashings(), v.SlashValidator)
    if err != nil {
        return nil, errors.Wrap(err, "could not process altair attester slashing")
    }
    st, err = altair.ProcessAttestationsNoVerifySignature(ctx, st, signedBeaconBlock)
    if err != nil {
        return nil, errors.Wrap(err, "could not process altair attestation")
    }
    if _, err := altair.ProcessDeposits(ctx, st, signedBeaconBlock.Block().Body().Deposits()); err != nil {
        return nil, errors.Wrap(err, "could not process altair deposit")
    }
    st, err = b.ProcessVoluntaryExits(ctx, st, signedBeaconBlock.Block().Body().VoluntaryExits())
    if err != nil {
        return nil, errors.Wrap(err, "could not process voluntary exits")
    }
    bailouts, err := signedBeaconBlock.Block().Body().BailOuts()
    if err != nil {
        return nil, errors.Wrap(err, "could not get bailouts from block body")
    }
    st, err = b.ProcessBailOuts(ctx, st, bailouts)
    if err != nil {
        return nil, errors.Wrap(err, "could not process bailouts")
    }
    return b.ProcessBLSToExecutionChanges(st, signedBeaconBlock)
}
```

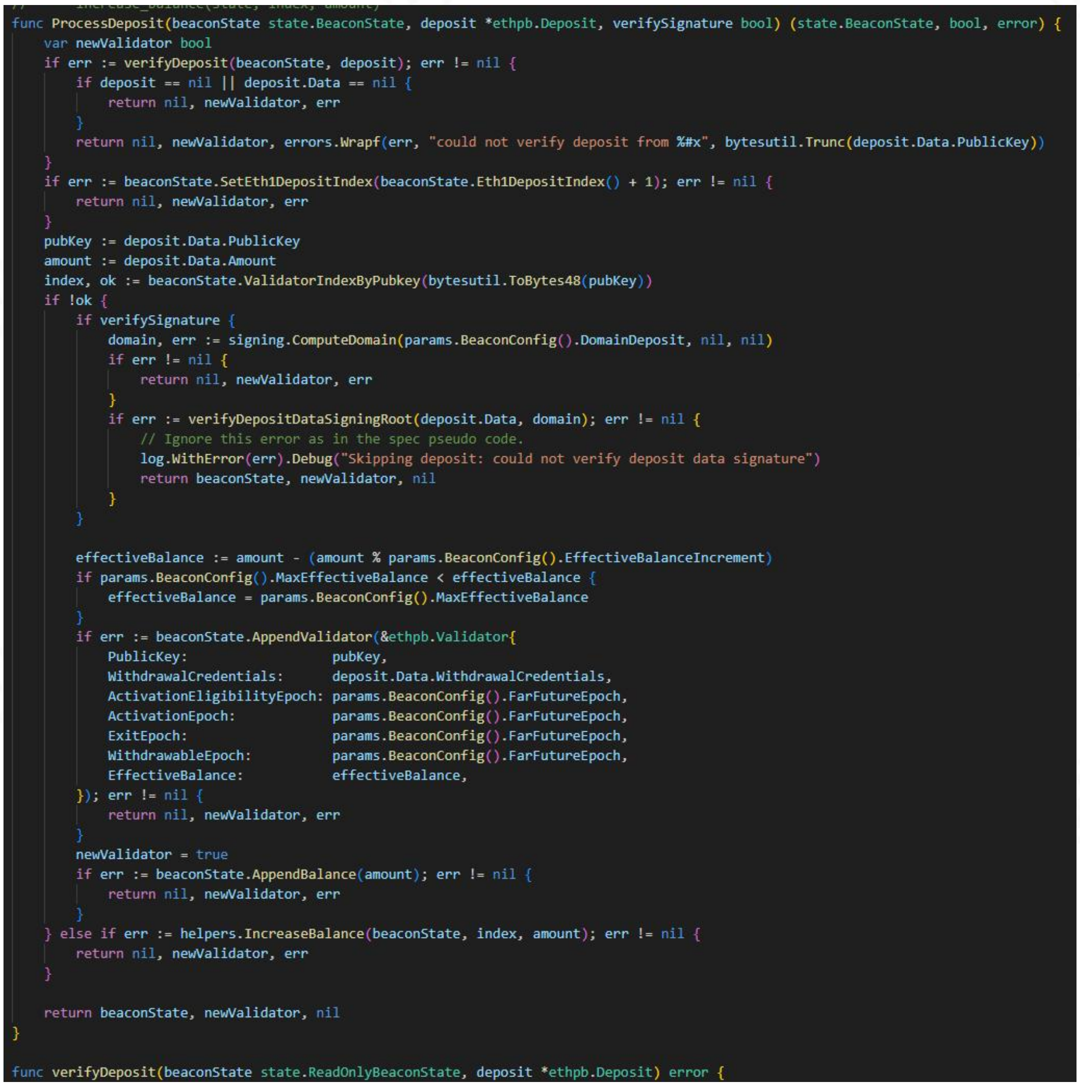
Figure 13 Source code of the *altairOperations* function

The *ProcessDeposit* function will verify whether the public key in the current deposit data exists in the Validator list. If it does not exist, it will create a new validator.



```go
func ProcessDeposits(
    ctx context.Context,
    beaconState state.BeaconState,
    deposits []*ethpb.Deposit,
) (state.BeaconState, error) {
    batchVerified, err := blocks.BatchVerifyDepositsSignatures(ctx, deposits)
    if err != nil {
        return nil, err
    }

    for _, deposit := range deposits {
        if deposit == nil || deposit.Data == nil {
            return nil, errors.New("got a nil deposit in block")
        }
        beaconState, err = ProcessDeposit(beaconState, deposit, batchVerified)
        if err != nil {
            return nil, errors.Wrapf(err, "could not process deposit from %#x", bytesutil.Trunc(deposit.Data.PublicKey))
        }
    }
    return beaconState, nil
}
```

Figure 14 Source code of the *ProcessDeposits* function

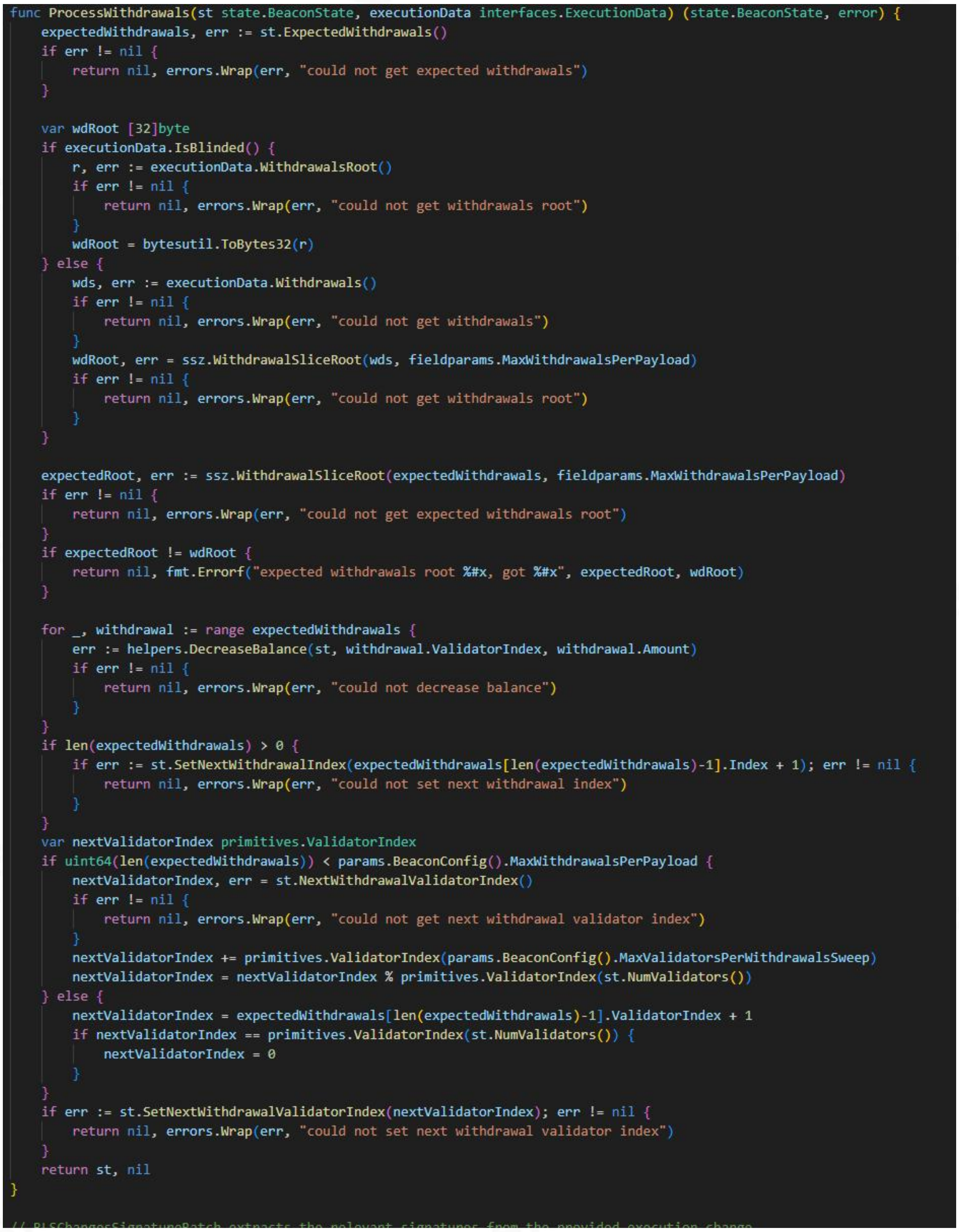


```go
func ProcessDeposit(beaconState state.BeaconState, deposit *ethpb.Deposit, verifySignature bool) (state.BeaconState, bool, error) {
    var newValidator bool
    if err := verifyDeposit(beaconState, deposit); err != nil {
        if deposit == nil || deposit.Data == nil {
            return nil, newValidator, err
        }
        return nil, newValidator, errors.Wrapf(err, "could not verify deposit from %#x", bytesutil.Trunc(deposit.Data.PublicKey))
    }
    if err := beaconState.SetEth1DepositIndex(beaconState.Eth1DepositIndex() + 1); err != nil {
        return nil, newValidator, err
    }
    pubKey := deposit.Data.PublicKey
    amount := deposit.Data.Amount
    index, ok := beaconState.ValidatorIndexByPubkey(bytesutil.ToBytes48(pubKey))
    if !ok {
        if verifySignature {
            domain, err := signing.ComputeDomain(params.BeaconConfig().DomainDeposit, nil, nil)
            if err != nil {
                return nil, newValidator, err
            }
            if err := verifyDepositDataSigningRoot(deposit.Data, domain); err != nil {
                // Ignore this error as in the spec pseudo code.
                log.WithError(err).Debug("Skipping deposit: could not verify deposit data signature")
                return beaconState, newValidator, nil
            }
        }

        effectiveBalance := amount - (amount % params.BeaconConfig().EffectiveBalanceIncrement)
        if params.BeaconConfig().MaxEffectiveBalance < effectiveBalance {
            effectiveBalance = params.BeaconConfig().MaxEffectiveBalance
        }
        if err := beaconState.AppendValidator(&ethpb.Validator{
            PublicKey:                  pubKey,
            WithdrawalCredentials:      deposit.Data.WithdrawalCredentials,
            ActivationEligibilityEpoch: params.BeaconConfig().FarFutureEpoch,
            ActivationEpoch:            params.BeaconConfig().FarFutureEpoch,
            ExitEpoch:                  params.BeaconConfig().FarFutureEpoch,
            WithdrawableEpoch:          params.BeaconConfig().FarFutureEpoch,
            EffectiveBalance:           effectiveBalance,
        }); err != nil {
            return nil, newValidator, err
        }
        newValidator = true
        if err := beaconState.AppendBalance(amount); err != nil {
            return nil, newValidator, err
        }
    } else if err := helpers.IncreaseBalance(beaconState, index, amount); err != nil {
        return nil, newValidator, err
    }

    return beaconState, newValidator, nil
}

func verifyDeposit(beaconState state.ReadOnlyBeaconState, deposit *ethpb.Deposit) error {
```

Figure 15 Source code of the *ProcessDeposit* function

3.3.3 Validator Withdraw

The *ProcessWithdrawals* function retrieves the corresponding list from the expected withdrawals and decreases the balances of the validators in that list.



```go
func ProcessWithdrawals(st state.BeaconState, executionData interfaces.ExecutionData) (state.BeaconState, error) {
    expectedWithdrawals, err := st.ExpectedWithdrawals()
    if err != nil {
        return nil, errors.Wrap(err, "could not get expected withdrawals")
    }

    var wdRoot [32]byte
    if executionData.IsBlinded() {
        r, err := executionData.WithdrawalsRoot()
        if err != nil {
            return nil, errors.Wrap(err, "could not get withdrawals root")
        }
        wdRoot = bytesutil.ToBytes32(r)
    } else {
        wds, err := executionData.Withdrawals()
        if err != nil {
            return nil, errors.Wrap(err, "could not get withdrawals")
        }
        wdRoot, err = ssz.WithdrawalSliceRoot(wds, fieldparams.MaxWithdrawalsPerPayload)
        if err != nil {
            return nil, errors.Wrap(err, "could not get withdrawals root")
        }
    }

    expectedRoot, err := ssz.WithdrawalSliceRoot(expectedWithdrawals, fieldparams.MaxWithdrawalsPerPayload)
    if err != nil {
        return nil, errors.Wrap(err, "could not get expected withdrawals root")
    }
    if expectedRoot != wdRoot {
        return nil, fmt.Errorf("expected withdrawals root %#x, got %#x", expectedRoot, wdRoot)
    }

    for _, withdrawal := range expectedWithdrawals {
        err := helpers.DecreaseBalance(st, withdrawal.ValidatorIndex, withdrawal.Amount)
        if err != nil {
            return nil, errors.Wrap(err, "could not decrease balance")
        }
    }
    if len(expectedWithdrawals) > 0 {
        if err := st.SetNextWithdrawalIndex(expectedWithdrawals[len(expectedWithdrawals)-1].Index + 1); err != nil {
            return nil, errors.Wrap(err, "could not set next withdrawal index")
        }
    }
    var nextValidatorIndex primitives.ValidatorIndex
    if uint64(len(expectedWithdrawals)) < params.BeaconConfig().MaxWithdrawalsPerPayload {
        nextValidatorIndex, err = st.NextWithdrawalValidatorIndex()
        if err != nil {
            return nil, errors.Wrap(err, "could not get next withdrawal validator index")
        }
        nextValidatorIndex += primitives.ValidatorIndex(params.BeaconConfig().MaxValidatorsPerWithdrawalsSweep)
        nextValidatorIndex = nextValidatorIndex % primitives.ValidatorIndex(st.NumValidators())
    } else {
        nextValidatorIndex = expectedWithdrawals[len(expectedWithdrawals)-1].ValidatorIndex + 1
        if nextValidatorIndex == primitives.ValidatorIndex(st.NumValidators()) {
            nextValidatorIndex = 0
        }
    }
    if err := st.SetNextWithdrawalValidatorIndex(nextValidatorIndex); err != nil {
        return nil, errors.Wrap(err, "could not set next withdrawal validator index")
    }
    return st, nil
}

// BLSChangesSignatureBatch extracts the relevant signatures from the provided execution change
```

Figure 16 Source code of the *ProcessWithdrawals* function

3.3.4 Asset Security

The Beacon Chain adopts the EIP-4895 standard, introducing a new operation type that allows for validator withdrawals directly from the Beacon Chain to the EVM at the system level. This approach does not generate transfers but unconditionally increases the specified user's balance, thereby mitigating various security issues.

This EIP provides a way for validator withdrawals made on the beacon chain to enter into the EVM. The architecture is "push"-based, rather than "pull"-based, where withdrawals are required to be processed in the execution layer as soon as they are dequeued from the consensus layer.

Withdrawals are represented as a new type of object in the execution payload – an "operation" – that separates the withdrawals feature from user-level transactions. This approach is more involved than the prior approach introducing a new transaction type but it cleanly separates this "system-level" operation from regular transactions. The separation simplifies testing (so facilitates security) by reducing interaction effects generated by mixing this system-level concern with user data.

Moreover, this approach is more complex than "pull"-based alternatives with respect to the core protocol but does provide tighter integration of a critical feature into the protocol itself.

3.3.5 Validator churnlimit

The beacon chain restricts the inflow and outflow rates of validators in each epoch based on the current number of active validators. Every 65,535 validators increase the current base churn limit by 1.

Additionally, Chronos has established a Deposit Plan at different times, which adjusts the churn limit beyond its own churn limit. When the activeValidatorDeposit exceeds the depositPlan, the validator outflow rate increases. When the activeValidatorDeposit is less than the depositPlan, the validator inflow rate increases.

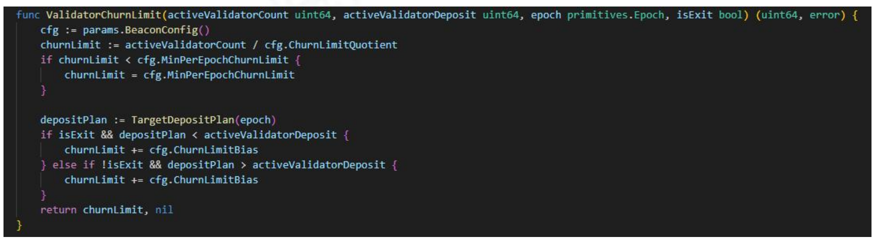

```go
func ValidatorChurnLimit(activeValidatorCount uint64, activeValidatorDeposit uint64, epoch primitives.Epoch, isExit bool) (uint64, error) {
    cfg := params.BeaconConfig()
    churnLimit := activeValidatorCount / cfg.ChurnLimitQuotient
    if churnLimit < cfg.MinPerEpochChurnLimit {
        churnLimit = cfg.MinPerEpochChurnLimit
    }

    depositPlan := TargetDepositPlan(epoch)
    if isExit && depositPlan < activeValidatorDeposit {
        churnLimit += cfg.ChurnLimitBias
    } else if !isExit && depositPlan > activeValidatorDeposit {
        churnLimit += cfg.ChurnLimitBias
    }
    return churnLimit, nil
}
```

Figure 17 Adjust churnlimit based on DepositPlan

This measure helps alleviate the pressure on consensus validation at different times, thereby increasing the security of the consensus.

3.4 Consensus security

3.4.1 Validators identity

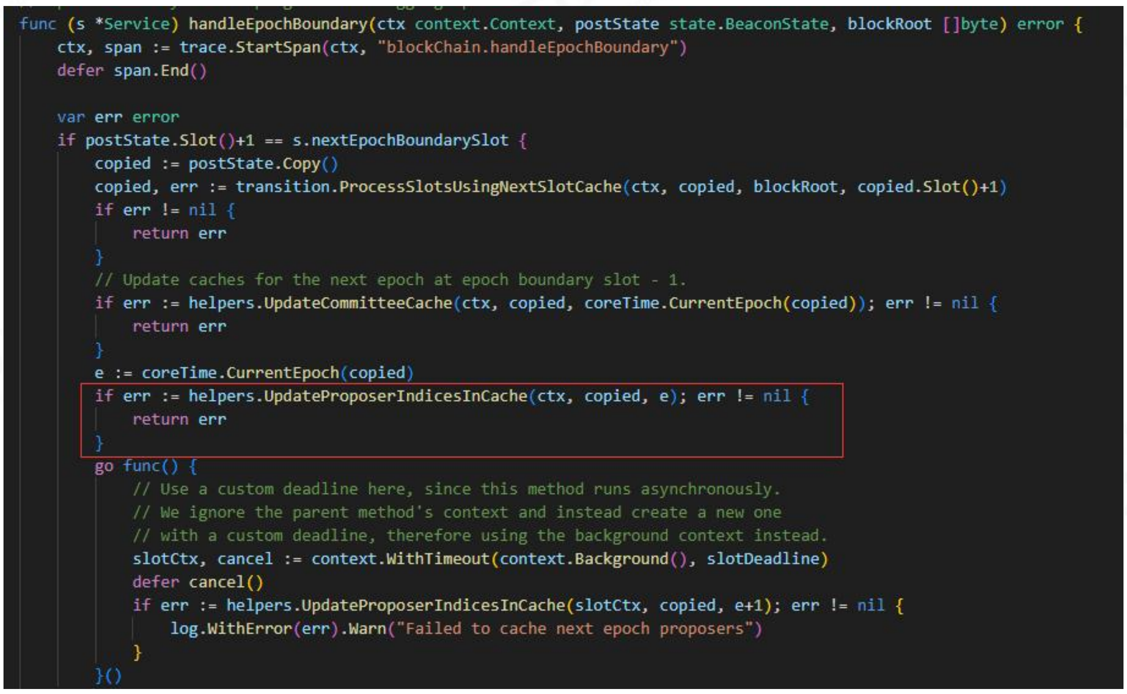
Validators usually have two identities: block proposer and attesters.

➢ Block proposer is a validator that has been pseudorandomly selected to build a block.

➢ Attesters usually vote on the blocks, These votes are recorded in the Beacon Chain and determine the head of the Beacon Chain. These Attesters will form committees of the same size based on the total number and vote to prove the slots they are responsible for on a committee basis, ensuring the integrity of consensus.

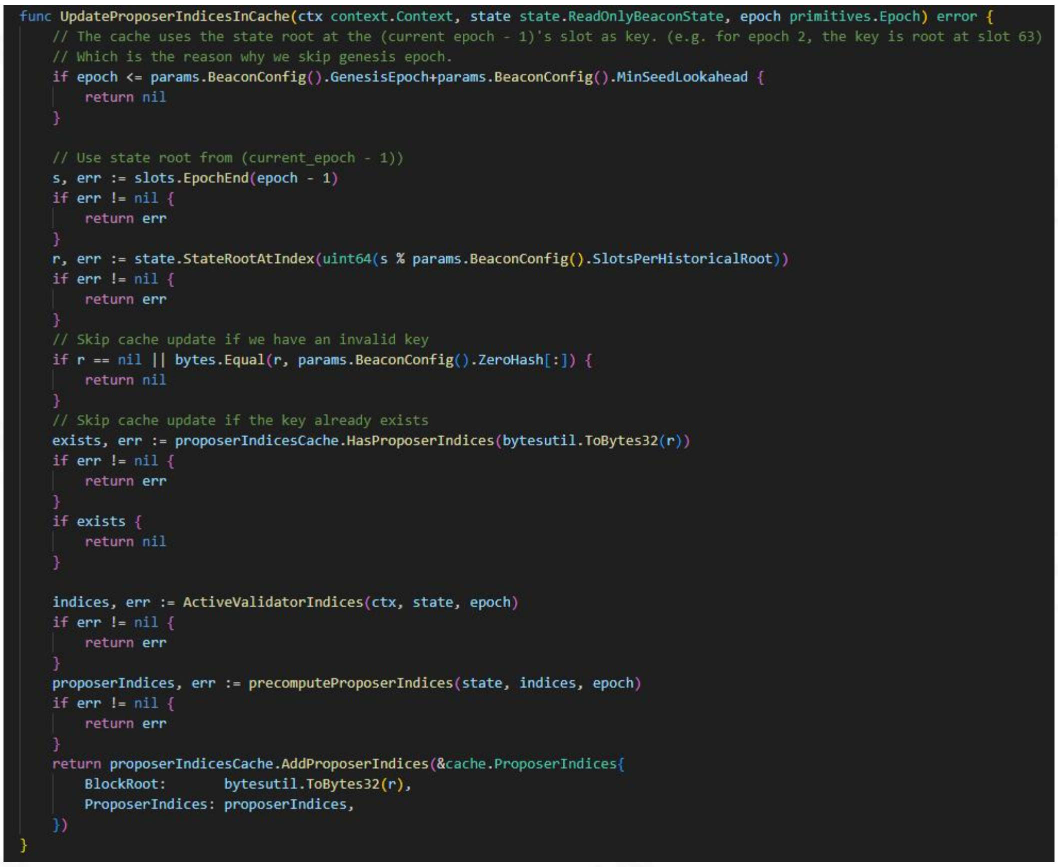
3.4.2 Block propose security

At the end of each epoch, the Beacon Chain randomly selects the proposers for the next epoch and the subsequent one from the current active validators and adds them to a cache. This process ensures a certain level of operational security for the Beacon Chain. The selected proposers are responsible for producing blocks in their assigned slots.



```go
func (s *Service) handleEpochBoundary(ctx context.Context, postState state.BeaconState, blockRoot []byte) error {
    ctx, span := trace.StartSpan(ctx, "blockChain.handleEpochBoundary")
    defer span.End()

    var err error
    if postState.Slot()+1 == s.nextEpochBoundarySlot {
        copied := postState.Copy()
        copied, err := transition.ProcessSlotsUsingNextSlotCache(ctx, copied, blockRoot, copied.Slot()+1)
        if err != nil {
            return err
        }
        // Update caches for the next epoch at epoch boundary slot - 1.
        if err := helpers.UpdateCommitteeCache(ctx, copied, coreTime.CurrentEpoch(copied)); err != nil {
            return err
        }
        e := coreTime.CurrentEpoch(copied)
        if err := helpers.UpdateProposerIndicesInCache(ctx, copied, e); err != nil {
            return err
        }
        go func() {
            // Use a custom deadline here, since this method runs asynchronously.
            // We ignore the parent method's context and instead create a new one
            // with a custom deadline, therefore using the background context instead.
            slotCtx, cancel := context.WithTimeout(context.Background(), slotDeadline)
            defer cancel()
            if err := helpers.UpdateProposerIndicesInCache(slotCtx, copied, e+1); err != nil {
                log.WithError(err).Warn("Failed to cache next epoch proposers")
            }
        }()
```

Figure 18 Source code of the *handleEpochBoundary* function



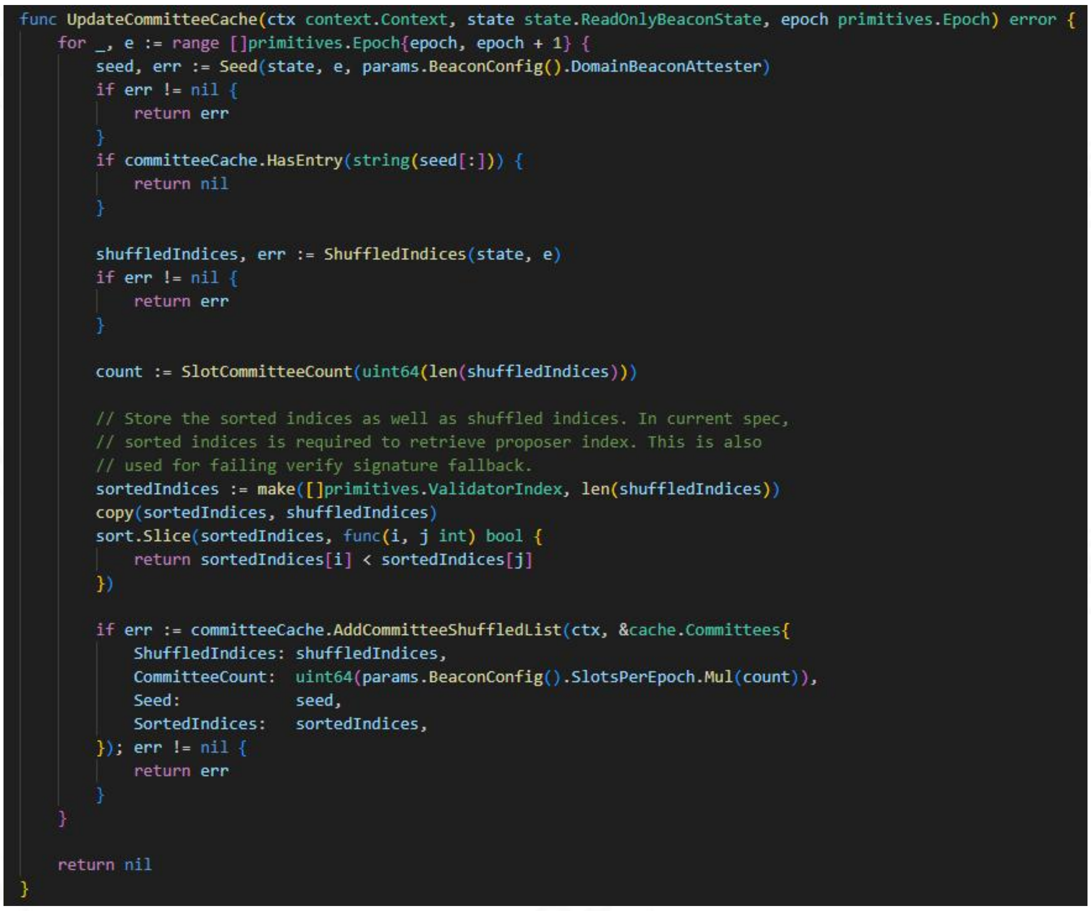
```go
func UpdateProposerIndicesInCache(ctx context.Context, state state.ReadOnlyBeaconState, epoch primitives.Epoch) error {
    // The cache uses the state root at the (current epoch - 1)'s slot as key. (e.g. for epoch 2, the key is root at slot 63)
    // Which is the reason why we skip genesis epoch.
    if epoch <= params.BeaconConfig().GenesisEpoch+params.BeaconConfig().MinSeedLookahead {
        return nil
    }

    // Use state root from (current_epoch - 1))
    s, err := slots.EpochEnd(epoch - 1)
    if err != nil {
        return err
    }
    r, err := state.StateRootAtIndex(uint64(s % params.BeaconConfig().SlotsPerHistoricalRoot))
    if err != nil {
        return err
    }
    // Skip cache update if we have an invalid key
    if r == nil || bytes.Equal(r, params.BeaconConfig().ZeroHash[:]) {
        return nil
    }
    // Skip cache update if the key already exists
    exists, err := proposerIndicesCache.HasProposerIndices(bytesutil.ToBytes32(r))
    if err != nil {
        return err
    }
    if exists {
        return nil
    }

    indices, err := ActiveValidatorIndices(ctx, state, epoch)
    if err != nil {
        return err
    }
    proposerIndices, err := precomputeProposerIndices(state, indices, epoch)
    if err != nil {
        return err
    }
    return proposerIndicesCache.AddProposerIndices(&cache.ProposerIndices{
        BlockRoot:       bytesutil.ToBytes32(r),
        ProposerIndices: proposerIndices,
    })
}
```

Figure 19 Source code of the *UpdateProposerIndicesInCache* function

If they successfully produce a block, they receive a block reward. However, if they fail to produce a block on time or if the block does not reach a finalized state, the proposer will face inactivity penalties, resulting in a deduction of their staked amount. Repeated failures to produce blocks can lead to a bailout, revoking the validator's status. Severe infractions, such as double proposals, can result in harsher slashing penalties. These measures ensure the stability and security of block production by the block proposers.

3.4.3 Attesters security

At the end of each epoch, the Beacon Chain elects committees for the next and subsequent epochs. The UpdateCommitteeCache function retrieves all currently active validators and divides them into multiple committees, storing these committees in a cache. Each committee is responsible for performing LMD GHOST votes in designated slots to attest to the blocks in those slots.



```
func UpdateCommitteeCache(ctx context.Context, state state.ReadOnlyBeaconState, epoch primitives.Epoch) error {
    for _, e := range []primitives.Epoch{epoch, epoch + 1} {
        seed, err := Seed(state, e, params.BeaconConfig().DomainBeaconAttester)
        if err != nil {
            return err
        }
        if committeeCache.HasEntry(string(seed[:])) {
            return nil
        }

        shuffledIndices, err := ShuffledIndices(state, e)
        if err != nil {
            return err
        }

        count := SlotCommitteeCount(uint64(len(shuffledIndices)))

        // Store the sorted indices as well as shuffled indices. In current spec,
        // sorted indices is required to retrieve proposer index. This is also
        // used for failing verify signature fallback.
        sortedIndices := make([]primitives.ValidatorIndex, len(shuffledIndices))
        copy(sortedIndices, shuffledIndices)
        sort.Slice(sortedIndices, func(i, j int) bool {
            return sortedIndices[i] < sortedIndices[j]
        })

        if err := committeeCache.AddCommitteeShuffledList(ctx, &cache.Committees{
            ShuffledIndices: shuffledIndices,
            CommitteeCount:  uint64(params.BeaconConfig().SlotsPerEpoch.Mul(count)),
            Seed:            seed,
            SortedIndices:   sortedIndices,
        }); err != nil {
            return err
        }
    }

    return nil
}
```

Figure 20 Source code of the *UpdateCommitteeCache* function

Successful attestations result in corresponding rewards. However, failing to attest on time incurs inactivity penalties, reducing the validator's staked amount. Repeated failure to attest can lead to a bailout, revoking the validator's status. More severe infractions, such as LMD GHOST double votes, may result in harsher slashing penalties. These measures ensure the stability and security of block attestations by the attesters.

3.4.4 Checkpoint security

In the first slot of each epoch, the block is typically referred to as a checkpoint. When validators initiate an LMD GHOST vote, they simultaneously cast a Casper FFG vote, which attests to the checkpoint of the current epoch. If over two-thirds of the validators approve this attestation, the current epoch is marked as justified. If the checkpoint of the subsequent epoch is also attested, the previously justified epoch is finalized, achieving a finalized state. This dual-epoch attestation process enhances security by requiring two epochs of validation to finalize a block, thereby ensuring a higher security threshold for block confirmation. Similarly, the implementation of FFG also plays a decisive role in the selection of the longest chain, thereby ensuring consensus security.

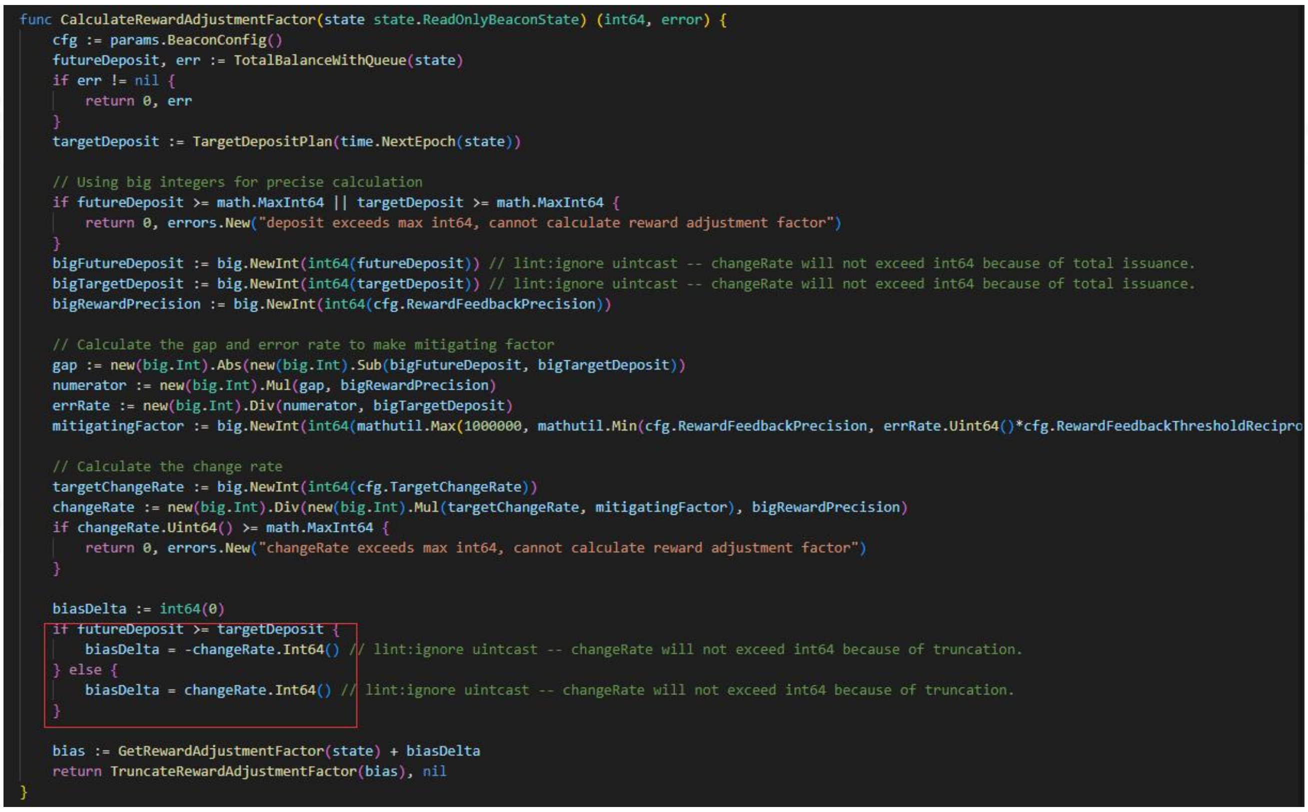
3.5 Reward model security

3.5.1 Consensus reward

In the next 10 years, Over protocol's current total issuance is set to 1 billion, and it is expected that 40% of this will be issued in the form of consensus rewards.

Year	Percentile from Total supply
1	2.0%
2	5.5%
3	7.5%
4	6.5%
5	5.0%
6	4.0%
7	3.0%
8	2.5%
9	2.3%
10	1.7%
Issuance total	40%

Based on this, Chronos will calculate a reward adjustment value according to the current DepositPlan, thereby adjusting the current consensus rewards. When the total effective balance of the current validators exceeds the DepositPlan, the consensus rewards will decrease, and the reduced consensus rewards will be added to the reserve fund for subsequent incentives. Conversely, when the total effective balance of validators is less than the DepositPlan, the consensus rewards will increase to incentivize users to become validators, and the increased rewards will be deducted from the accumulated reserve fund. These incentives can effectively protect the integrity of the consensus.

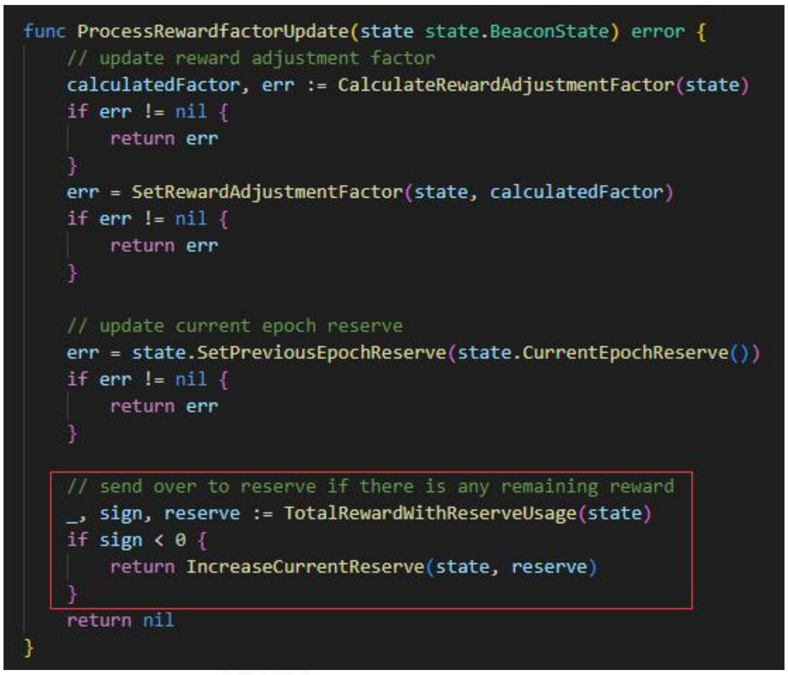


```go
func CalculateRewardAdjustmentFactor(state state.ReadOnlyBeaconState) (int64, error) {
    cfg := params.BeaconConfig()
    futureDeposit, err := TotalBalanceWithQueue(state)
    if err != nil {
        return 0, err
    }
    targetDeposit := TargetDepositPlan(time.NextEpoch(state))

    // Using big integers for precise calculation
    if futureDeposit >= math.MaxInt64 || targetDeposit >= math.MaxInt64 {
        return 0, errors.New("deposit exceeds max int64, cannot calculate reward adjustment factor")
    }
    bigFutureDeposit := big.NewInt(int64(futureDeposit)) // lint:ignore uintcast -- changeRate will not exceed int64 because of total issuance.
    bigTargetDeposit := big.NewInt(int64(targetDeposit)) // lint:ignore uintcast -- changeRate will not exceed int64 because of total issuance.
    bigRewardPrecision := big.NewInt(int64(cfg.RewardFeedbackPrecision))

    // Calculate the gap and error rate to make mitigating factor
    gap := new(big.Int).Abs(new(big.Int).Sub(bigFutureDeposit, bigTargetDeposit))
    numerator := new(big.Int).Mul(gap, bigRewardPrecision)
    errRate := new(big.Int).Div(numerator, bigTargetDeposit)
    mitigatingFactor := big.NewInt(int64(mathutil.Max(1000000, mathutil.Min(cfg.RewardFeedbackPrecision, errRate.Uint64()*cfg.RewardFeedbackThresholdRecipro

    // Calculate the change rate
    targetChangeRate := big.NewInt(int64(cfg.TargetChangeRate))
    changeRate := new(big.Int).Div(new(big.Int).Mul(targetChangeRate, mitigatingFactor), bigRewardPrecision)
    if changeRate.Uint64() >= math.MaxInt64 {
        return 0, errors.New("changeRate exceeds max int64, cannot calculate reward adjustment factor")
    }

    biasDelta := int64(0)
    if futureDeposit >= targetDeposit {
        biasDelta = -changeRate.Int64() // lint:ignore uintcast -- changeRate will not exceed int64 because of truncation.
    } else {
        biasDelta = changeRate.Int64() // lint:ignore uintcast -- changeRate will not exceed int64 because of truncation.
    }

    bias := GetRewardAdjustmentFactor(state) + biasDelta
    return TruncateRewardAdjustmentFactor(bias), nil
}
```

Figure 21 Adjust rewards based on DepositPlan

3.5.2 Reserve security

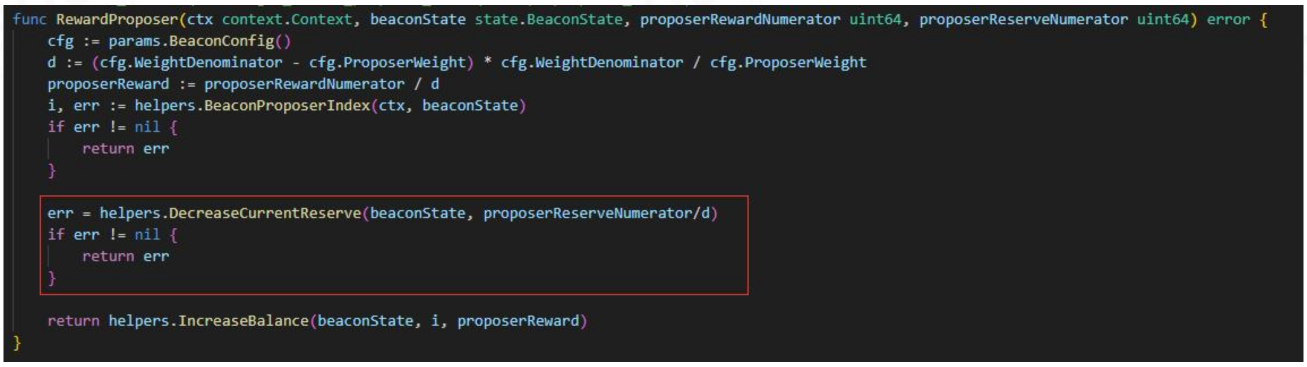
The adjustment value deducted or added to the base reward will be increased or deducted from the reserve fund. The existence of the reserve fund can effectively incentivize users when deposit is low, and avoid excessive growth of validator when deposit is excessive, thus enhancing the sustainability of the project.



```go
func ProcessRewardfactorUpdate(state state.BeaconState) error {
	// update reward adjustment factor
	calculatedFactor, err := CalculateRewardAdjustmentFactor(state)
	if err != nil {
		return err
	}
	err = SetRewardAdjustmentFactor(state, calculatedFactor)
	if err != nil {
		return err
	}

	// update current epoch reserve
	err = state.SetPreviousEpochReserve(state.CurrentEpochReserve())
	if err != nil {
		return err
	}

	// send over to reserve if there is any remaining reward
	_, sign, reserve := TotalRewardWithReserveUsage(state)
	if sign < 0 {
		return IncreaseCurrentReserve(state, reserve)
	}
	return nil
}
```

Figure 22 Increase reserves

The rewards granted to the proposer will also be issued from the reserve fund. It is worth noting that even if the current reserve fund is insufficient for issuance, the proposer's rewards will still be distributed as usual. This will cause a certain degree of increase in consensus rewards.

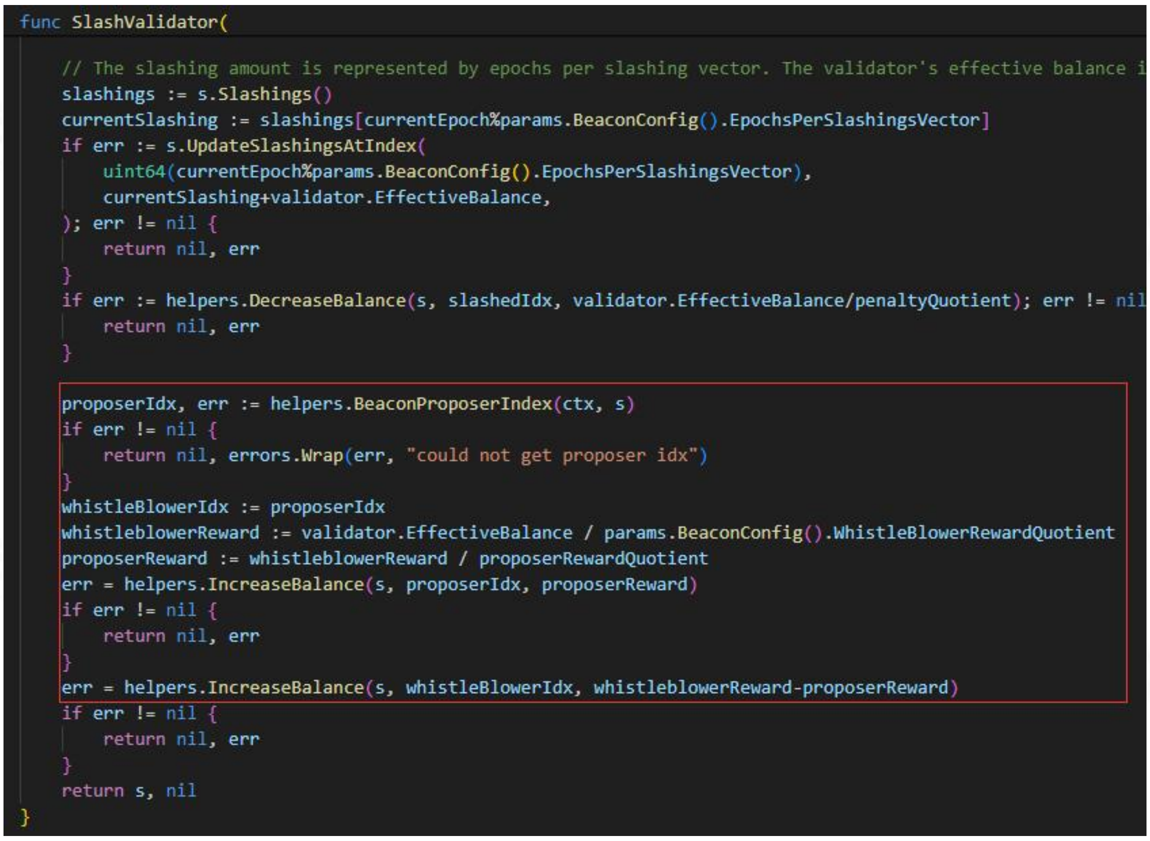


```go
func RewardProposer(ctx context.Context, beaconState state.BeaconState, proposerRewardNumerator uint64, proposerReserveNumerator uint64) error {
	cfg := params.BeaconConfig()
	d := (cfg.WeightDenominator - cfg.ProposerWeight) * cfg.WeightDenominator / cfg.ProposerWeight
	proposerReward := proposerRewardNumerator / d
	i, err := helpers.BeaconProposerIndex(ctx, beaconState)
	if err != nil {
		return err
	}

	err = helpers.DecreaseCurrentReserve(beaconState, proposerReserveNumerator/d)
	if err != nil {
		return err
	}

	return helpers.IncreaseBalance(beaconState, i, proposerReward)
}
```

Figure 23 Reduce reserves to pay reward

3.5.3 Whistleblower rewards security

After a successful report, the whistleblower will receive a reward, which is determined by the EffectiveBalance of the reported validator and the reporting reward coefficient. The existence of the reporter reward incentivizes validators within the system to actively search for misbehaving validators, thereby stabilizing the operation of the system.



```go
func SlashValidator(

    // The slashing amount is represented by epochs per slashing vector. The validator's effective balance i
    slashings := s.Slashings()
    currentSlashing := slashings[currentEpoch%params.BeaconConfig().EpochsPerSlashingsVector]
    if err := s.UpdateSlashingsAtIndex(
        uint64(currentEpoch%params.BeaconConfig().EpochsPerSlashingsVector),
        currentSlashing+validator.EffectiveBalance,
    ); err != nil {
        return nil, err
    }
    if err := helpers.DecreaseBalance(s, slashedIdx, validator.EffectiveBalance/penaltyQuotient); err != nil
        return nil, err
    }

    proposerIdx, err := helpers.BeaconProposerIndex(ctx, s)
    if err != nil {
        return nil, errors.Wrap(err, "could not get proposer idx")
    }
    whistleBlowerIdx := proposerIdx
    whistleblowerReward := validator.EffectiveBalance / params.BeaconConfig().WhistleBlowerRewardQuotient
    proposerReward := whistleblowerReward / proposerRewardQuotient
    err = helpers.IncreaseBalance(s, proposerIdx, proposerReward)
    if err != nil {
        return nil, err
    }
    err = helpers.IncreaseBalance(s, whistleBlowerIdx, whistleblowerReward-proposerReward)
    if err != nil {
        return nil, err
    }
    return s, nil
}
```

Figure 24 Issue rewards to whistleBlower

3.6 Penalties module security

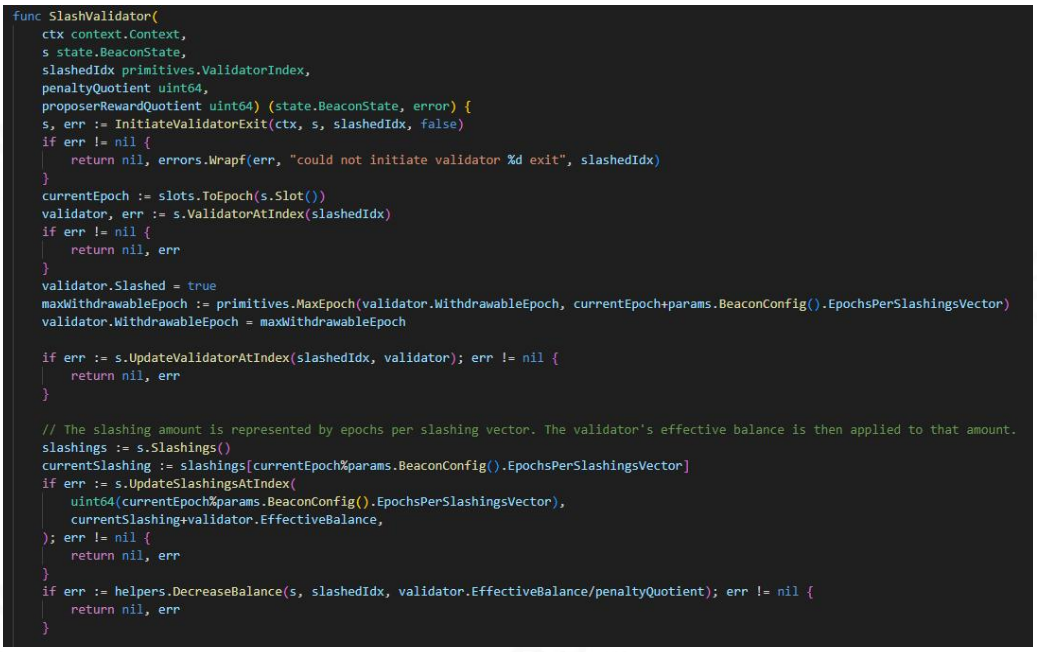
The Penalties in Chronos are divided into three types: Slash, Inactivity, and Bailout. Among them, Inactivity and Bailout will be enforced after accumulating a certain Score, while Slash will punish the corresponding validator after a successful report.

3.6.1 Slash penalty

Slash penalty is typically generated by reporting. Validators will be slashed if they exhibit the following four conditions: double proposal, LMD GHOST double vote, FFG surround vote, and FFG double vote.

- Double proposal is a proposer proposing more than one block for their assigned slot.
- LMD GHOST double vote is a validator attesting to two different Beacon Chain heads for their assigned slot.
- Surround vote is a validator casting an FFG vote that surrounds or is surrounded by a previous FFG vote they made.
- FFG double vote is a validator casting 2 FFG votes for any two targets at the same epoch.This can happen during a fork.

These penalty measures can effectively increase the risk of validators misbehaving during block production and attestation, thus deterring the occurrence of forks and similar issues.

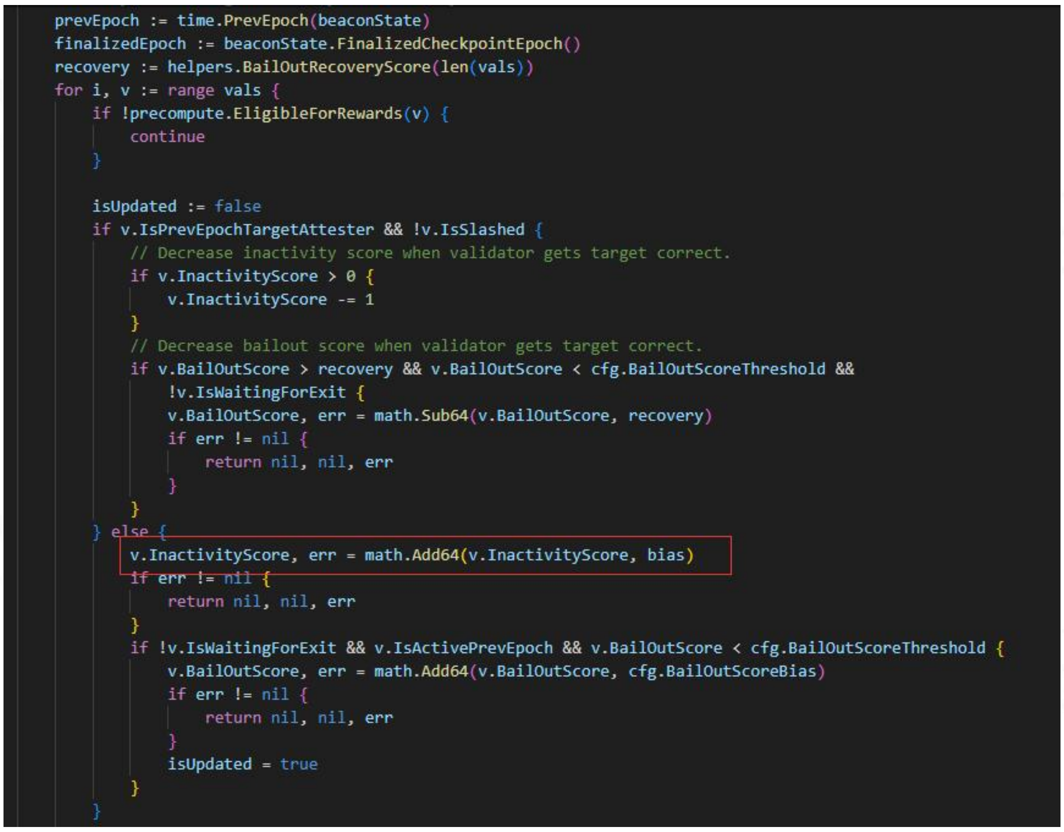


```go
func SlashValidator(
	ctx context.Context,
	s state.BeaconState,
	slashedIdx primitives.ValidatorIndex,
	penaltyQuotient uint64,
	proposerRewardQuotient uint64) (state.BeaconState, error) {
	s, err := InitiateValidatorExit(ctx, s, slashedIdx, false)
	if err != nil {
		return nil, errors.Wrapf(err, "could not initiate validator %d exit", slashedIdx)
	}
	currentEpoch := slots.ToEpoch(s.Slot())
	validator, err := s.ValidatorAtIndex(slashedIdx)
	if err != nil {
		return nil, err
	}
	validator.Slashed = true
	maxWithdrawableEpoch := primitives.MaxEpoch(validator.WithdrawableEpoch, currentEpoch+params.BeaconConfig().EpochsPerSlashingsVector)
	validator.WithdrawableEpoch = maxWithdrawableEpoch

	if err := s.UpdateValidatorAtIndex(slashedIdx, validator); err != nil {
		return nil, err
	}

	// The slashing amount is represented by epochs per slashing vector. The validator's effective balance is then applied to that amount.
	slashings := s.Slashings()
	currentSlashing := slashings[currentEpoch%params.BeaconConfig().EpochsPerSlashingsVector]
	if err := s.UpdateSlashingsAtIndex(
		uint64(currentEpoch%params.BeaconConfig().EpochsPerSlashingsVector),
		currentSlashing+validator.EffectiveBalance,
	); err != nil {
		return nil, err
	}
	if err := helpers.DecreaseBalance(s, slashedIdx, validator.EffectiveBalance/penaltyQuotient); err != nil {
		return nil, err
	}
```

Figure 25 Source code of the *SlashValidator* function

3.6.2 Inactivity penalty

Validators receive rewards through attestations and proposals. Conversely, if they fail to attest or propose during their corresponding slots, it will result in Inactivity penalties. Each time a validator fails to participate in consensus, the system accumulates an InactivityScore (which can be used to determine inactive status) and applies penalties subsequently.

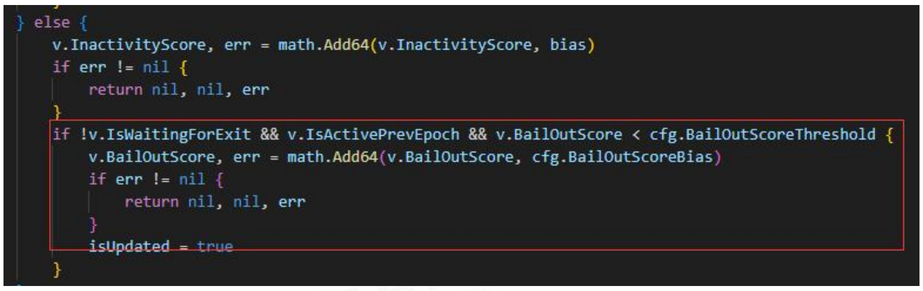


```
prevEpoch := time.PrevEpoch(beaconState)
finalizedEpoch := beaconState.FinalizedCheckpointEpoch()
recovery := helpers.BailOutRecoveryScore(len(vals))
for i, v := range vals {
    if !precompute.EligibleForRewards(v) {
        continue
    }

    isUpdated := false
    if v.IsPrevEpochTargetAttester && !v.IsSlashed {
        // Decrease inactivity score when validator gets target correct.
        if v.InactivityScore > 0 {
            v.InactivityScore -= 1
        }
        // Decrease bailout score when validator gets target correct.
        if v.BailOutScore > recovery && v.BailOutScore < cfg.BailOutScoreThreshold &&
            !v.IsWaitingForExit {
            v.BailOutScore, err = math.Sub64(v.BailOutScore, recovery)
            if err != nil {
                return nil, nil, err
            }
        }
    } else {
        v.InactivityScore, err = math.Add64(v.InactivityScore, bias)
        if err != nil {
            return nil, nil, err
        }
        if !v.IsWaitingForExit && v.IsActivePrevEpoch && v.BailOutScore < cfg.BailOutScoreThreshold {
            v.BailOutScore, err = math.Add64(v.BailOutScore, cfg.BailOutScoreBias)
            if err != nil {
                return nil, nil, err
            }
            isUpdated = true
        }
    }
```

Figure 26 Increase inactivity score

3.6.3 Bailout penalty

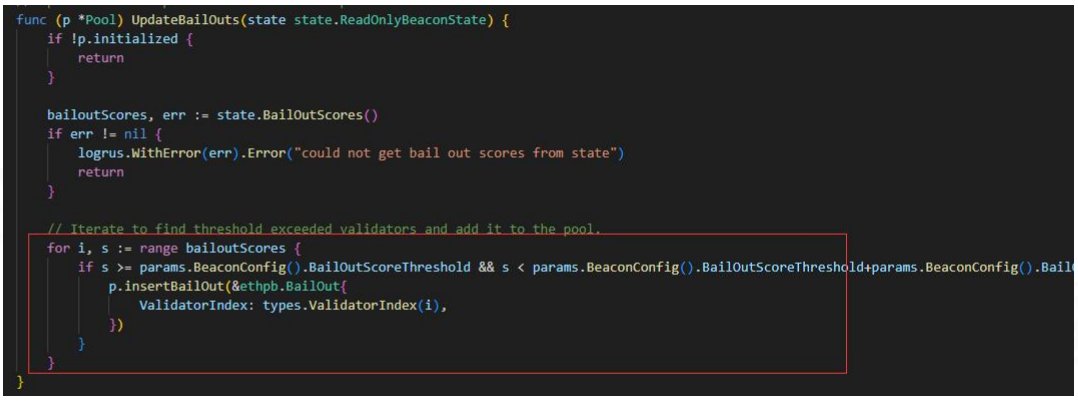
Chronos introduced a new mechanism called Bailout, which synchronously accumulates a BailoutScore when InactivityScore is accumulated.



```
} else {
    v.InactivityScore, err = math.Add64(v.InactivityScore, bias)
    if err != nil {
        return nil, nil, err
    }
    if !v.IsWaitingForExit && v.IsActivePrevEpoch && v.BailOutScore < cfg.BailOutScoreThreshold {
        v.BailOutScore, err = math.Add64(v.BailOutScore, cfg.BailOutScoreBias)
        if err != nil {
            return nil, nil, err
        }
        isUpdated = true
    }
```

Figure 27 Increase bailout score

When the BailoutScore reaches a certain threshold, the validator will be automatically added to the bailoutpool and subsequently ejected from the system. This design not only protects the verifier's deposit balance, but also ensures the stability of the system.



```go
func (p *Pool) UpdateBailOuts(state state.ReadOnlyBeaconState) {
    if !p.initialized {
        return
    }

    bailoutScores, err := state.BailOutScores()
    if err != nil {
        logrus.WithError(err).Error("could not get bail out scores from state")
        return
    }

    // Iterate to find threshold exceeded validators and add it to the pool.
    for i, s := range bailoutScores {
        if s >= params.BeaconConfig().BailOutScoreThreshold && s < params.BeaconConfig().BailOutScoreThreshold+params.BeaconConfig().Bail
            p.insertBailOut(&ethpb.BailOut{
                ValidatorIndex: types.ValidatorIndex(i),
            })
        }
    }
}
```

Figure 28 Add validators that exceed the threshold to the bailout pool

4 Historical Vulnerability Detection

Chronos is a fork version of Prysm Ver.v0.4.7 released in July 2023. Through this audit, we found that Chronos contains vulnerabilities present in higher versions of Prysm, although the team claims to have fixed them using patches from version 0.4.8. We still recommend the team allocate sufficient security resources to continuously monitor and fix vulnerabilities and issues exposed by Prysm to prevent Chronos nodes from being compromised.

4.1 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in blockchain.

4.2 About Beosin

Beosin is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions. Beosin has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, Beosin has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website
https://www.beosin.com

Telegram
https://t.me/beosin

Twitter
https://twitter.com/Beosin_com

Email
service@beosin.com