Flare Songbird / Coreth

Security Review Update: June 10, 2025

Reviewer: balthasar@gofyeo.com

June 2025
Version 1.0

Presented by:
FYEO Inc.
PO Box 147044
Lakewood CO 80214
United States

Flare Security Review Update

New security issues, 4

After the development team implemented the latest updates, FYEO conducted a review of the modifications. The primary goal of this evaluation was to ensure the continued robustness of the network's security features, safeguarding the network's integrity and maintaining the overall robustness of the codebase.

General Updates:

Coreth:

The code changes introduce network-specific configurations and features for Flare networks (Flare, Songbird, Coston). This includes custom gas limit handling, activation times for network upgrades, and specialized contract interactions. New core functionality is implemented for inflation minting through a daemon system, state connector operations for attestations, and governance mechanisms for address updates.

Transaction processing is enhanced with prioritized contract support and custom fee handling. The changes also introduce RPC batch processing limits with size validations and new error codes. Network-specific rules are applied to import/export transactions, and chain identification logic is extended to support new Flare network IDs throughout the system.

The updates include stricter validation of gas limits across network phases, enhanced RPC batch processing with size constraints to prevent resource exhaustion, and modified signature verification supporting both Avalanche and Ethereum formats. The implementation enforces governance contract interactions through authorized coinbase signaling and introduces a daemon controlled inflation mechanism with chain specific minting caps to limit economic impact.

Avalanche:

The changes introduce support for several new networks (Flare, Songbird, Coston, Costwo, and their “local” variants) throughout the configuration, genesis, and VM layers. New genesis files and parameters are added for each network, and the code that previously assumed only “mainnet” and “local” now recognizes these additional chain IDs. Network-specific fee and staking parameters are routed dynamically, so that each chain can have its own Tx fees, minimum/maximum stakes, and upgrade timelines instead of relying on static defaults. Several test files are updated to replace references to the old Fuji testnet with Songbird or Flare equivalents, and new tests ensure that each network’s genesis and VM behaviors are validated.

Versioning and upgrade logic are extended to handle these new chains: application prefixes (e.g. “flare/” vs. “avalanche/”) are set based on network ID, and various hard-coded upgrade

timestamps (for phases like Cortina, Durango, Banff, etc.) now include entries for the new networks. The staking and permissionless transaction verification logic is modified to pull its parameters from the network-specific “inflation settings,” allowing each chain to enforce its own minimum/maximum stakes, delegation fees, and time-based restrictions. Finally, reward calculations are disabled (always returning zero) for these new networks, and EVM-style signature recovery is added in the AVM so that transactions can be verified using either the native or Ethereum-style “signed message” flow once the appropriate upgrade is active.

By centralizing staking and fee parameters in network-specific “inflation settings”, the updates make security checks time and chain dependent, meaning consensus relies on each node’s clock to enforce minimum stakes, durations, and fee thresholds—potentially letting clock skew alter economic protections. Signature verification is also expanded to accept Ethereum signed messages once a chain reaches a certain upgrade. Upgrade timestamps for features like BLS enforcement (Durango) or EVM signatures (Banff) are hard-coded per network, so any misconfiguration could allow pre-upgrade attacks or inadvertently bypass critical signature checks.

For this part of the code it was noted that BLS signature enforcement introduced via the Durango update is disabled, since the update time has been set far into the future. This also implies that the Durango features relying on BLS signatures are not activated.

AtomicDaemonAndMint may not rever in all cases

Finding ID: FYEO-FLARE-01
Severity: Medium
Status: Acknowledged

Description

In the function `atomicDaemonAndMint(evm EVMCaller, log log.Logger)` within `core/daemon.go`, if the `daemon(evm)` call returns an error (`daemonErr`), the code merely logs a warning and does not revert any state changes or abort further processing. As a result, any partial state changes made by the daemon call (e.g., updates to the EVM context or side effects prior to the error) are not rolled back, potentially leaving the EVM in an inconsistent state.

Proof of Issue

File name: core/daemon.go
Line numbers: ~160–175

```
func atomicDaemonAndMint(evm EVMCaller, log log.Logger) {
    // Call the daemon
    daemonSnapshot, mintRequest, daemonErr := daemon(evm)
    // If no error...
    if daemonErr == nil {
        // time to mint
        if mintError := mint(evm, mintRequest); mintError != nil {
            log.Warn("Error minting inflation request", "error", mintError)
            // Revert to snapshot to unwind daemon state transition
            evm.DaemonRevertToSnapshot(daemonSnapshot)
        }
    } else {
        log.Warn("Daemon error", "error", daemonErr)
        // ← No revert or other action if daemonErr != nil
    }
}
```

Severity and Impact Summary

The `daemon()` call likely involves a `DaemonCall` into a precompiled or external contract that may modify internal EVM state (e.g., storage, balances, or context). If `daemonErr` occurs after some modifications, failing to revert to the saved `daemonSnapshot` leaves those partial changes applied. This can corrupt the execution environment, leading to incorrect balances or storage values that break consensus or contract logic.

Recommendation

Make sure this behaviour is implemented according to specifications and document the reasoning.

Missing gas checks

Finding ID: FYEO-FLARE-02
Severity: Medium
Status: Remediated

Description

In Songbird's ApricotPhase1 (and earlier) code paths, there is no enforcement that the block header's `GasLimit` matches the fixed `ApricotPhase1GasLimit`. The verification logic only checks for specific Songbird transition and ApricotPhase5 limits, leaving ApricotPhase1 completely unverified when running under `IsSongbirdCode()`. As a result, a miner can propose a block with any arbitrary gas limit during ApricotPhase1, and it will be accepted by verifiers, bypassing the fixed-limit rule.

Proof of Issue

File name: consensus/dummy/consensus.go
Line number: ~118

```
if config.IsSongbirdCode() {
    // Verify that the gas limit is correct for the current phase
    if config.IsSongbirdTransition(header.Time) {
        if header.GasLimit != params.SgbTransitionGasLimit {
            return fmt.Errorf(
                "expected gas limit to be %d in SgbTransition but found %d",
                params.SgbTransitionGasLimit, header.GasLimit,
            )
        }
    } else if config.IsApricotPhase5(header.Time) {
        if header.GasLimit != params.SgbApricotPhase5GasLimit {
            return fmt.Errorf(
                "expected gas limit to be %d in ApricotPhase5 but found %d",
                params.SgbApricotPhase5GasLimit, header.GasLimit,
            )
        }
    }
    // ← Missing: no check for ApricotPhase1 under IsSongbirdCode()
} else {
    ...
}
```

File name: plugin/evm/block_verification.go
Line number: ~113

```
if rules.IsSongbirdCode {
    if rules.IsSongbirdTransition {
        if ethHeader.GasLimit != params.SgbTransitionGasLimit {
            return fmt.Errorf(
                "expected gas limit to be %d in sgb transition but got %d",
                params.SgbTransitionGasLimit, ethHeader.GasLimit,
            )
        }
    } else if rules.IsApricotPhase5 {
```

```
        if ethHeader.GasLimit != params.SgbApricotPhase5GasLimit {
            return fmt.Errorf(
                "expected gas limit to be %d in apricot phase 5 but got %d",
                params.SgbApricotPhase5GasLimit, ethHeader.GasLimit,
            )
        }
    }
    // ← Missing: no enforcement when IsApricotPhase1 under IsSongbirdCode()
} else {
    ...
}
```

Because the `IsApricotPhase1` branch is only in the “else” (non-Songbird) path, Songbird’s ApricotPhase1 blocks are never checked against `params.ApricotPhase1GasLimit`.

Severity and Impact Summary

Without enforcement, a malicious miner can propose a block during ApricotPhase1 with an arbitrary and potentially excessively large gas limit. This subverts the fixed-gas-limit guarantees that ApricotPhase1 is supposed to impose. All Songbird validators and light clients running a code path where `IsSongbirdCode()` is true (after the Songbird hard fork) but before ApricotPhase5 do not reject blocks with an incorrect ApricotPhase1 gas limit. Because the code for ApricotPhase1 is missing entirely under `IsSongbirdCode()`, every block in that window is vulnerable.

Recommendation

Update the header verification and block verification routines to explicitly enforce the ApricotPhase1 gas limit whenever `config.IsSongbirdCode()` (or `rules.IsSongbirdCode`) and `config.IsApricotPhase1(header.Time)` (or `rules.IsApricotPhase1`) both return true.

TransitionDb concern

Finding ID: FYEO-FLARE-03
Severity: Low
Status: Remediated

Description

In `TransitionDb` (in `core/state_transition.go`), after a successful EVM call (`vmerr == nil`), the code checks only whether `chainID` is non-nil and whether the chain is Songbird (`isSongbird`). It invokes `handleSongbirdTransitionDbContracts` if `isSongbird` is true; otherwise it unconditionally calls `handleFlareTransitionDbContracts`. As a result, **all non-Songbird chains** (including Costwo, Coston, local testnets, etc.) are treated exactly like Flare. This lumps every chain other than Songbird into the "Flare" branch, even when they have different governance or state-connector requirements.

Proof of Issue

File name: core/state_transition.go
Line numbers: ~464–480

```
    ret, st.gasRemaining, vmerr = st.evm.Call(sender, st.to(), msg.Data,
st.gasRemaining, msg.Value)
    if vmerr == nil && chainID != nil {
        if isSongbird {
            handleSongbirdTransitionDbContracts(st, chainID, timestamp, msg,
ret)
        } else {
            handleFlareTransitionDbContracts(st, chainID, timestamp, msg, ret)
        }
    }
```

Severity and Impact Summary

Non-Flare chains (Costwo, Coston, local environments) will erroneously run Flare's state-connector or governance hooks, potentially invoking the wrong daemon calls or "finalise" logic when their block.coinbase matches the Flare-specific signal address.

If Coston or Costwo transactions inadvertently match the Flare selectors, the code will attempt `st.SetGovernanceAddress` or `st.UpdateInitialAirdropAddress` under `handleFlareTransitionDbContracts`. Because `GetGovernanceSettingIsActivatedAndCalled`/`GetInitialAirdropChangeIsActivatedAndCalled` use Flare activation times and Flare addresses, these calls may silently fail or, worse, modify state incorrectly—leading to inconsistent state across nodes.

Developers running local or staging networks may see Flare-specific hooks firing when they expect nothing.

Recommendation

Introduce an `isFlare` boolean (already returned by `stateTransitionVariants.GetValue(chainID)(st)`) and use it to branch to `handleFlareTransitionDbContracts` only when `isFlare == true`. This ensures that Costwo, Coston, and local chains will not accidentally invoke Flare callbacks.

CODE CLARITY

Finding ID: FYEO-FLARE-04
Severity: **Informational**
Status: **Remediated**

Description

The code in `core/daemon.go` and `core/governance_settings.go` contains several clarity and correctness issues.

Proof of Issue

File name: core/daemon.go
Line number: 94

```
// Call the method
daemonSnapshot, daemonRet, _, daemonErr := evm.DaemonCall(
    vm.AccountRef(daemonContract),
    daemonContract,
    GetDaemonSelector(evm.GetBlockTime()),
    GetDaemonGasMultiplier(evm.GetBlockTime())*evm.GetGasLimit())  // ← multiplication here
```

Because both `GetDaemonGasMultiplier(...)` and `evm.GetGasLimit()` are `uint64`, their product can overflow before being passed as the `gas` argument to `DaemonCall`.

File name: core/daemon.go
Line number: 136

```
func mint(evm EVMCaller, mintRequest *big.Int) error {
    max := GetMaximumMintRequest(evm.GetChainID(), evm.GetBlockTime())
    if mintRequest.Cmp(big.NewInt(0)) > 0 &&
        mintRequest.Cmp(max) <= 0 {

evm.AddBalance(common.HexToAddress(GetDaemonContractAddr(evm.GetBlockTime())),
mintRequest)
    } else if mintRequest.Cmp(max) > 0 {
        return &ErrMaxMintExceeded{…}
    } else if mintRequest.Cmp(big.NewInt(0)) < 0 {
        return &ErrMintNegative{}
    }
    // ← if mintRequest == 0, we end up here with no comment or special handling
    return nil
}
```

There is no explicit comment or branch for `mintRequest == 0`, making it unclear whether zero should succeed silently or be treated as an error/no-op.

File name: core/governance_settings.go
Line numbers: 130–144

```
func (st *StateTransition) UpdateInitialAirdropAddress(chainID *big.Int,
timestamp uint64) error {
```

```
    coinbaseSignal := GetInitialAirdropChangeCoinbaseSignalAddr(chainID,
timestamp)
    originalCoinbase := st.evm.Context.Coinbase
    defer func() {
        st.evm.Context.Coinbase = originalCoinbase
    }()
    st.evm.Context.Coinbase = coinbaseSignal
    _, _, _, err := st.evm.DaemonCall(vm.AccountRef(coinbaseSignal), st.to(),
st.msg.Data, st.evm.Context.GasLimit)
    if err != nil {
        return err
    }
    initialAirdropAddress := GetInitialAirdropContractAddress(chainID,
timestamp)
    targetAidropAddress := GetTargetAirdropContractAddress(chainID, timestamp)
    airdropBalance := st.state.GetBalance(initialAirdropAddress)
    st.state.SubBalance(initialAirdropAddress, airdropBalance)
    st.state.AddBalance(targetAidropAddress, airdropBalance)
    return nil
}

func (st *StateTransition) UpdateDistributionAddress(chainID *big.Int,
timestamp uint64) error {
    coinbaseSignal := GetDistributionChangeCoinbaseSignalAddr(chainID,
timestamp)
    originalCoinbase := st.evm.Context.Coinbase
    defer func() {
        st.evm.Context.Coinbase = originalCoinbase
    }()
    st.evm.Context.Coinbase = coinbaseSignal
    _, _, _, err := st.evm.DaemonCall(vm.AccountRef(coinbaseSignal), st.to(),
st.msg.Data, st.evm.Context.GasLimit)
    if err != nil {
        return err
    }
    distributionAddress := GetDistributionContractAddress(chainID, timestamp)
    targetDistributionAddress := GetTargetDistributionContractAddress(chainID,
timestamp)
    distributionBalance := st.state.GetBalance(distributionAddress)
    st.state.SubBalance(distributionAddress, distributionBalance)
    st.state.AddBalance(targetDistributionAddress, distributionBalance)
    return nil
}
```

Neither function checks `if initialAirdropAddress == targetAirdropAddress { ... }` (similarly for distribution), so if the "target" equals the "source," funds are subtracted and re-added to the same account.

Severity and Impact Summary

If `GetDaemonGasMultiplier(...) * evm.GetGasLimit()` exceeds `2^64-1`, it wraps around silently. The daemon call may then receive an unexpectedly small (or zero) gas limit, causing unintended behavior (e.g., out-of-gas, denial-of-service, or unexpected reverts). While this might be rare—depending on the actual ranges of `GetDaemonGasMultiplier` and `evm.GetGasLimit()`, it is a correctness bug that can lead to unpredictable execution.

Because zero is neither explicitly allowed nor rejected, readers may misunderstand intended behavior. Although a zero mint request simply becomes a no-op, callers might expect an error or a log entry. At best this is confusing; at worst it allows zero-value calls to slip through unnoticed, potentially masking higher-level logic errors.

If `GetInitialAirdropContractAddress(...)` equals `GetTargetAirdropContractAddress(...)`, the code subtracts and then re-adds the same balance. While this is functionally a no-op, it is misleading and potentially harmful if future code assumes the source and target are always distinct.

Collectively, these issues degrade code clarity and can lead to subtle bugs, misconfigurations, or unexpected behavior in production.

Recommendation

Guard against overflows. Clarify zero-value behavior in `mint()`. Add source/target checks in `UpdateInitialAirdropAddress` and `UpdateDistributionAddress`. By centralizing these clarity checks and small-scale safeguards, the code becomes more robust, easier to audit, and less prone to subtle overflows or misconfigurations.

Commit Hash Reference:

For transparency and reference, the security review was conducted on a specific commit hash. The commit hash for the reviewed version is as follows:

Avalanchego: 5b99eccc323cd013e813d1e03faedb0a690922f2
Coreth: 4cd443cfe57a1a36b91b8029eae58599b246d2ab

Conclusion:

In conclusion, the security aspects of the Flare network remain robust and unaffected by the recent updates. Users can confidently interact with the network, assured that their assets are well-protected. The commitment to security exhibited by the development team is commendable, and we appreciate the ongoing efforts to prioritize the safeguarding of network users.