About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

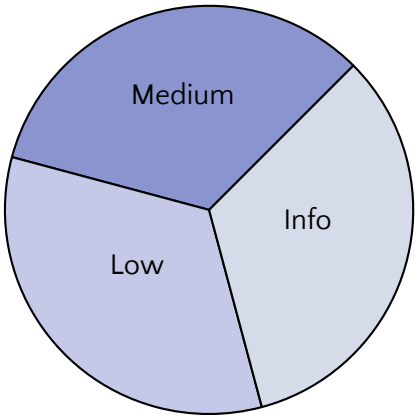
Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1.3 Results

During our assessment on the scoped Synapse Sanguine contracts, we discovered three findings. No critical issues were found. One was of medium impact, one was of low impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	1



2 Introduction

2.1 About Synapse Sanguine

Synapse is a universal interoperability protocol that enables secure cross-chain communication. Synapse connects blockchains by offering an extensible cross-chain messaging protocol that supports assets, smart contract calls, and more.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality

standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Synapse Sanguine Contracts

Repository https://github.com/synapsecns/sanguine

Version sanguine: `b9a87f6c7bf51335dcbc83092f8135be091cda36`

Programs

- Destination.sol
- BondingManager.sol
- AgentManager.sol
- LightManager.sol
- Number.sol
- Tips.sol
- GasData.sol
- Header.sol
- Request.sol

- TypeCasts.sol
- MerkleMath.sol
- MerkleTree.sol
- Structures.sol
- Message.sol
- Attestation.sol
- Receipt.sol
- ByteString.sol
- MemView.sol
- State.sol
- Snapshot.sol
- BaseMessage.sol
- Constants.sol
- Errors.sol
- MultiCallable.sol
- Version.sol
- AgentSecured.sol
- MessagingBase.sol
- LightInbox.sol
- Inbox.sol
- StatementInbox.sol
- Summit.sol
- TestClient.sol
- MessageRecipient.sol
- PingPongClient.sol
- BaseClient.sol
- Origin.sol

3 Detailed Findings

3.1 Potentially uninitialized implementation contracts

- **Target**: Project Wide
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

Implementation contracts designed to be called by a proxy should always be initialized to prevent potential takeovers.

Impact

If an implementation contract is not initialized, an attacker could be able to initialize it and perform a `selfdestruct`, deleting the implementation contract and causing a denial of service.

Recommendations

Ensure the implementation contract is always initialized.

Following official OpenZeppelin documentation, this can be accomplished by defining a constructor on the contract:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

For more information, refer to these OpenZeppelin documents.

Remediation

This issue has been acknowledged by Synapse.

3.2 Centralization risk of `GasData`

- **Target**: GasOracle
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Gas oracles play a significant role in the project to determine the prices and costs of cross-chain messaging. The fields affected are described below:

```
/// | Position | Field | Type | Bytes | Description |
/// | ———————— | ————————— | ———— | ——— |
    ——————————————————————————————————————————— |
/// | (012..010] | gasPrice | uint16 | 2 | Gas price for the chain (in Wei
    per gas unit) |
/// | (010..008] | dataPrice | uint16 | 2 | Calldata price (in Wei per
    byte of content) |
/// | (008..006] | execBuffer | uint16 | 2 | Tx fee safety buffer for
    message execution (in Wei) |
/// | (006..004] | amortAttCost | uint16 | 2 | Amortized cost for
    attestation submission (in Wei) |
/// | (004..002] | etherPrice | uint16 | 2 | Chain's Ether price / Mainnet
    Ether price (in BWAD) |
/// | (002..000] | markup | uint16 | 2 | Markup for the message execution
    (in BWAD) |
```

These fields are updated on the local chain by using the local chain's `GasOracle` through the `setGasData` function.

```
/// @notice MVP function to set the gas data for the given domain.
function setGasData(
    uint32 domain,
    uint256 gasPrice,
    uint256 dataPrice,
    uint256 execBuffer,
    uint256 amortAttCost,
    uint256 etherPrice,
    uint256 markup
) external onlyOwner {
    GasData updatedGasData = GasDataLib.encodeGasData({
```

```
            gasPrice_: NumberLib.compress(gasPrice),
            dataPrice_: NumberLib.compress(dataPrice),
            execBuffer_: NumberLib.compress(execBuffer),
            amortAttCost_: NumberLib.compress(amortAttCost),
            etherPrice_: NumberLib.compress(etherPrice),
            markup_: NumberLib.compress(markup)
        });
        if (GasData.unwrap(updatedGasData)
        ≠ GasData.unwrap(_gasData[domain])) {
            _setGasData(domain, updatedGasData);
        }
    }
```

Impact

In the case of a security compromise of the `onlyOwner` addresses' private key, on-chain gas data fields could be altered to halt or even steal funds from users and operators.

Recommendations

A first-level defense would be using a multi-key wallet to guarantee more than a single point of failure, making such attacks unlikely to succeed — in the long term, it is recommended to architect the design in a way that maximizes the decentralization of gas data.

Remediation

The Synapse team acknowledged this finding and replied with said comment:

> `setGasData` would be deprecated over time as we move to statistics being collected fully onchain, but until then the owner will be setting the 'uncollected' statistics. For some of the fields it will be a combination of both. For instance, `amortAttCost` is cost of submitting an attestation amortized among N messages that will be executed using it as proof. So the owner/governance will be able to set N, as some information how much gas/calldata is required for posting an attestation on average. Using collected gas/calldata price, amortized cost will be derived. More extreme example is markup (which defines how much the Executors are paid for successful execution vs just trying to correctly execute), which will be fully set by owner/governance.

3.3 Safety checks missing

- **Target**: BondingManager
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: Low
- **Impact**: Informational

Description

In the Synapse protocol, agents are responsible for attesting to the validity of cross-chain messages, as well as guarding the network from malicious activities. The action of slashing would occur when either type of agent(notary or guard) was to perform an action which would be considered malicious or illegitimate (e.g. allowing forged payloads to be passed over to the other chains).

The `remoteSlashAgent` from BondingManager allows the Destination contract to slash agents from the network. Currently however, it does not check whether the `msgOrigin`, which points to the domain of where the message originated, is different than the `domain` or `block.chainid`.

```
function remoteSlashAgent(uint32 msgOrigin, uint256 proofMaturity,
    uint32 domain, address agent, address prover)
    external
    returns (bytes4 magicValue)
{
    // Only destination can pass Manager Messages
    if (msg.sender ≠ destination) revert CallerNotDestination();
    // Check that merkle proof is mature enough
    // TODO: separate constant for slashing optimistic period
    if (proofMaturity < BONDING_OPTIMISTIC_PERIOD)
    revert SlashAgentOptimisticPeriod();
    // TODO: do we need to save this?
    msgOrigin;

    // @audit assure that msgOrigin ≠ domain or ≠ block.chainid?

    // Slash agent and notify local AgentSecured contracts
    _slashAgent(domain, agent, prover);
    // Magic value to return is selector of the called function
    return this.remoteSlashAgent.selector;
}
```

Impact

The impact of this potential issue is limited due to the fact that the function can only be called by the Destination contract. It could have led to using the `remoteSlashAgent` for slashing the agents from the same chain, potentially bypassing any constraints that other functions might have when calling `_slashAgent`.

Recommendations

Add a check to ensure that the `msgOrigin` is different than the `domain` or `block.chainid`.

```
function remoteSlashAgent(uint32 msgOrigin, uint256 proofMaturity,
    uint32 domain, address agent, address prover)
    external
    returns (bytes4 magicValue)
{
    // Only destination can pass Manager Messages
    if (msg.sender ≠ destination) revert CallerNotDestination();
    // ...

    if (msgOrigin == domain || msgOrigin == block.chainid) revert
        InvalidMsgOrigin();
    // ...
}
```

Remediation

This issue has been acknowledged by Synapse.

4 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1 Module: AgentManager.sol

Function: `openDispute(uint32 guardIndex, uint32 notaryIndex)`

Allows Inbox to open a dispute between a guard and a notary.

Inputs

- `guardIndex`
  - **Control**: Fully controlled by Inbox.
  - **Constraints**: Assured that the agent is not in dispute.
  - **Impact**: The index of the guard in the agent Merkle tree.
- `notaryIndex`
  - **Control**: Fully controlled by Inbox.
  - **Constraints**: Assured that the agent is not in dispute.
  - **Impact**: The index of the notary in the agent Merkle tree.

Branches and code coverage (including function calls)

Intended branches

- Should open a dispute between the guard and the notary.
  ☑ Test coverage
- Mark the guard and the notary as in dispute through the `agentDispute` mapping.
  ☑ Test coverage
- Notify all other chains that a dispute has been opened between the guard and the notary.
  ☑ Test coverage
- Assumes that the Inbox (caller) does prior checks as to the validity of the guard and the notary indices.

☑ Test coverage

- Add the dispute to the disputes array.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the inbox to call this function.
  ☑ Negative test

Function: `resolveStuckDispute(uint32 domain, address slashedAgent)`

Allows owner to resolve a stuck dispute.

Inputs

- `domain`
  - **Control**: Fully controlled by owner.
  - **Constraints**: Checked to be same domain as the agent's domain.
  - **Impact**: The domain of the agent that is being slashed.
- `slashedAgent`
  - **Control**: Fully controlled by owner.
  - **Constraints**: Checked to be in dispute.
  - **Impact**: The agent that is being slashed.

Branches and code coverage (including function calls)

Intended branches

- Should slash the agent from the dispute, should it be stuck.
  ☑ Test coverage

Negative behavior

- No one other than the owner can call this function.
  ☑ Negative test
- Should revert if the agent is not in dispute.
  ☑ Negative test
- Should not allow resolving a dispute that has not passed `FRESH_DATA_TIMEOUT` + `snapRootTime`.
  ☑ Negative test
- Should not allow slashing an agent whose domain does not match the given domain.
  ☑ Negative test

Function: `slashAgent(uint32 domain, address agent, address prover)`

Allows Inbox to slash an agent, if their fraud was proven.

Inputs

- `domain`
  - **Control**: Fully controlled by Inbox.
  - **Constraints**: Checked to be the same domain as the agent's domain.
  - **Impact**: The domain of the agent that is being slashed.
- `agent`
  - **Control**: Fully controlled by Inbox.
  - **Constraints**: Checked to be in dispute.
  - **Impact**: The agent that is being slashed.
- `prover`
  - **Control**: Fully controlled by Inbox.
  - **Constraints**: None.
  - **Impact**: The address that initially provided fraud proof.

Branches and code coverage (including function calls)

Intended branches

- Should slash the agent if their fraud was proven.
  ☑ Test coverage
- Should revert if the agent is not in dispute.
  ☑ Test coverage
- Should revert if the agent is not in the same domain as the given domain.
  ☑ Test coverage
- Call the `afterAgentSlashed` hook.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the inbox to call this function.
  ☑ Negative test

4.2 Module: AgentSecured.sol

Function: `openDispute(uint32 guardIndex, uint32 notaryIndex)`

Used by the agent manager to open a dispute between a guard and a notary.

Inputs

- `guardIndex`
  - **Control**: Fully controlled by the agent manager.
  - **Constraints**: None.
  - **Impact**: The agent (presumably a guard) with this index will be marked as disputed.
- `notaryIndex`
  - **Control**: Fully controlled by the agent manager.
  - **Constraints**: None.
  - **Impact**: The agent (presumably a notary) with this index will be marked as disputed.

Branches and code coverage (including function calls)

Intended branches

- Update the `_disputes` mapping for both the `guardIndex` and `notaryIndex`.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than `agentManager`.
  ☑ Negative test
- Should not allow opening a dispute for an agent that is already in a dispute.
  ☐ Negative test
- Should not allow for `guardIndex` and `notaryIndex` to be the same.
  ☐ Negative test

Function: `resolveDispute(uint32 slashedIndex, uint32 honestIndex)`

Used by agent manager to resolve a dispute between a guard and a notary.

Inputs

- `slashedIndex`
  - **Control**: Fully controlled by the agent manager.
  - **Constraints**: None.
  - **Impact**: The agent with this index will be marked as slashed.
- `honestIndex`
  - **Control**: Fully controlled by the agent manager.
  - **Constraints**: Checked to be nonzero.
  - **Impact**: The agent with this index will have their dispute flag removed.

Branches and code coverage (including function calls)

Intended branches

- Update the `_disputes` mapping for `slashedIndex` to `DisputeFlag.Slashed`.
  ☑ Test coverage
- Delete the `_disputes` mapping for `honestIndex`.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than `agentManager`.
  ☑ Negative test
- Should not allow resolving a dispute for an agent that is not in a dispute.
  ☐ Negative test
- Should not allow for `slashedIndex` and `honestIndex` to be the same.
  ☐ Negative test

4.3 Module: BondingManager.sol

Function: `addAgent(uint32 domain, address agent, byte[32][] proof)`

Allows adding a new agent for the domain.

Inputs

- `domain`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Should domain be equal to `SYNAPSE_DOMAIN`, revert.
  - **Impact**: The domain of the agent.
- `agent`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Depend on the current status of the agent within the Merkle tree.
  - **Impact**: The address of the agent.
- `proof`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Should be a valid proof for the current status of the agent within the Merkle tree.
  - **Impact**: The proof of the agent's status.

Branches and code coverage (including function calls)

Intended branches

- Should add a new agent to the Merkle tree if all conditions are met.
  ☑ Test coverage
- Push the agent to `_agents`.
  ☑ Test coverage
- Push the agent to `_domainAgents[domain]`.
  ☑ Test coverage
- Assure that one agent can only be added once for a given domain.
  ☐ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  ☑ Negative test
- Should not be able to add an agent to `SYNAPSE_DOMAIN`.
  ☑ Negative test

Function: `completeSlashing(uint32 domain, address agent, byte[32][] proof)`

Allows completing the slashing of the agent bond.

Inputs

- `domain`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Assured that it matches the agent's domain.
  - **Impact**: The domain of the agent.
- `agent`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Assured it is a legitimate agent.
  - **Impact**: The address of the agent.
- `proof`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Should be a valid proof for the current status of the agent within the Merkle tree.
  - **Impact**: Proof of the agent's status.

Branches and code coverage (including function calls)

Intended branches

- Should allow anyone to call this function, assuming they have valid proof.
  ☐ Test coverage
- Should complete the slashing of the agent if all conditions are met.
  ☑ Test coverage
- Should update the agent's status to Slashed.
  ☑ Test coverage

Negative behavior

- Should not perform slashing if the agent is not fraudulent or if the proof is invalid.
  ☑ Negative test
- Should not be able to slash an agent that is not active or unstaking.
  ☑ Negative test
- Should not be able to slash an agent that is not in the given domain.
  ☑ Negative test

Function: `completeUnstaking(uint32 domain, address agent, byte[32][] proof)`

Allows completing the unstaking of the agent bond.

Inputs

- `domain`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Assured that it matches the agent's domain.
  - **Impact**: The domain of the agent.
- `agent`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Assured it is a legitimate agent.
  - **Impact**: The address of the agent.
- `proof`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Should be a valid proof for the current status of the agent within the Merkle tree.
  - **Impact**: Proof of the agent's status.

Branches and code coverage (including function calls)

Intended branches

- Should complete the unstaking of the agent if all conditions are met.
  ☑ Test coverage
- Should update the agent's status to Resting.
  ☑ Test coverage

Negative behavior

- Should not allow unstaking unless previously initiated.
  ☑ Negative test
- Should not allow anyone other than the owner to call this function.
  ☑ Negative test
- Should not be able to complete the unstaking of an agent that is not unstaking.
  ☑ Negative test
- Complete the rest of the intended TO-DOs.
  ☐ Negative test

Function: `initialize(address origin_, address destination_, address inbox_, address summit_)`

Intializes the contract with the given addresses.

Inputs

- `origin_`
  - **Control**: Fully controlled by the deployer.
  - **Constraints**: None.
  - **Impact**: The address of the origin contract.
- `destination_`
  - **Control**: Fully controlled by the deployer.
  - **Constraints**: None.
  - **Impact**: The address of the destination contract.
- `inbox_`
  - **Control**: Fully controlled by the deployer.
  - **Constraints**: None.
  - **Impact**: The address of the inbox contract.
- `summit_`
  - **Control**: Fully controlled by the deployer.
  - **Constraints**: None.

  - **Impact**: The address of the summit contract.

Branches and code coverage (including function calls)

Intended branches

- Should call all underlying initializers (`AgentManager`, `Ownable`, `MessagingBase`).
  ☑ Test coverage
- Set summit to `summit_`.
  ☑ Test coverage
- Append a zero address to `_agents`.
  ☑ Test coverage
- Set the owner to `msg.sender`.
  ☑ Test coverage

Negative behavior

- Should not be callable multiple times.
  ☑ Negative test

Function: `initiateUnstaking(uint32 domain, address agent, byte[32][] proof)`

Allows initiating the unstaking of the agent bond.

Inputs

- `domain`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Assured that it matches the agent's domain.
  - **Impact**: The domain of the agent.
- `agent`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Assured it is a legitimate agent.
  - **Impact**: The address of the agent.
- `proof`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: Should be a valid proof for the current status of the agent within the Merkle tree.
  - **Impact**: Proof of the agent's status.

Branches and code coverage (including function calls)

Intended branches

- Should initiate the unstaking of the agent if all conditions are met.
  ☑ Test coverage
- Should update the agent's status to Unstaking.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the owner to call this function.
  ☑ Negative test
- Should not be able to initiate the unstaking of an agent that is not active.
  ☑ Negative test
- Should not be able to initiate the unstaking of an agent that is not in the given domain.
  ☑ Negative test

Function: `remoteSlashAgent(uint32 msgOrigin, uint256 proofMaturity, uint32 domain, address agent, address prover)`

Allows slashing an agent on the Synapse chain when their fraud is proven on the remote chain.

Inputs

- `msgOrigin`
  - **Control**: Fully controlled by the Destination contract.
  - **Constraints**: None.
  - **Impact**: The domain of the agent.
- `proofMaturity`
  - **Control**: Fully controlled by the Destination contract.
  - **Constraints**: Should be less than the `BONDING_OPTIMISTIC_PERIOD`.
  - **Impact**: The proof maturity of the agent.
- `domain`
  - **Control**: Fully controlled by the Destination contract.
  - **Constraints**: Checked that the agent is in the given domain.
  - **Impact**: The domain of the agent.
- `agent`
  - **Control**: Fully controlled by the Destination contract.
  - **Constraints**: Checked that the agent is in the given domain as well as that

it can be slashed.
  - **Impact**: The address of the agent.
- `prover`
  - **Control**: Fully controlled by the Destination contract.
  - **Constraints**: None.
  - **Impact**: The address of the prover.

Branches and code coverage (including function calls)

Intended branches

- Should slash the agent if all conditions are met.
  ☑ Test coverage

Negative behavior

- Should not allow the `msgOrigin` to be the same domain as the current chain. Currently not performed.
  ☐ Negative test
- Should not allow anyone other than the destination to call this function.
  ☑ Negative test
- Should not be able to slash an agent that is not active or unstaking.
  ☑ Negative test
- Should not be able to slash an agent that is not in the given domain.
  ☑ Negative test
- Should not be able to slash an agent that is not mature enough.
  ☑ Negative test

Function: `withdrawTips(address recipient, uint32 origin_, uint256 amount)`

Allows withdrawing locked base message tips from requested domain Origin to the recipient.

Inputs

- `recipient`
  - **Control**: Fully controlled by the Summit contract.
  - **Constraints**: None.
  - **Impact**: The address of the recipient.
- `origin_`
  - **Control**: Fully controlled by the Summit contract.

  - **Constraints**: None.
  - **Impact**: The domain of the origin.
- `amount`
  - **Control**: Fully controlled by the Summit contract.
  - **Constraints**: None.
  - **Impact**: The amount of tips to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Should withdraw tips if all conditions are met.
  ☑ Test coverage
- Should call local Origin to withdraw tips if the origin is local.
  ☑ Test coverage
- Increase the balance of the recipient by the amount.
  ☑ Test coverage
- Decrease the balance of the origin by the amount.
  ☑ Test coverage
- Should send a manager message to remote LightManager to handle the withdrawal if the origin is remote.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the Summit to call this function.
  ☑ Negative test

Function call analysis

- `origin.withdrawTips(recipient, amount)`
  - **What is controllable?** `recipient` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means there is not enough balance to withdraw.
- `origin.sendManagerMessage(origin_, BONDING_OPTIMISTIC_PERIOOD, abi.encodeWithSelector(InterfaceLightManager.remoteWithdrawTips.selector, recipient, amount))`
  - **What is controllable?** `origin_`, `recipient`, and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns `messageNonce` and `messageHash`, but these are currently not checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Means there is not enough balance to withdraw.

4.4 Module: Destination.sol

Function: `acceptAttestation(uint32 notaryIndex, uint256 sigIndex, byte[] attPayload, byte[32] agentRoot, ChainGas[] snapGas)`

Accepts an attestation on the destination chain.

Inputs

- `notaryIndex`
  - **Control**: Full.
  - **Constraints**: `onlyInbox`, verified to be a notary in the list.
  - **Impact**: Index of notary.
- `sigIndex`
  - **Control**: Full.
  - **Constraints**: `onlyInbox`, verified to be a real index in sig list.
  - **Impact**: Index of sig in `sigList`.
- `attPayload`
  - **Control**: Full.
  - **Constraints**: `onlyInbox`, verified to be a valid attestation.
  - **Impact**: Payload of the attestation.
- `agentRoot`
  - **Control**: Full.
  - **Constraints**: `onlyInbox` and `agentRoom` from the `agentManager`.
  - **Impact**: Root of the agent Merkle tree.
- `snapGas`
  - **Control**: Full.
  - **Constraints**: `onlyInbox`.
  - **Impact**: Gas data.

Branches and code coverage (including function calls)

Intended branches

- Agent root is updated if different and optimistic period is over.
  ☑ Test coverage

Negative behavior

- Notary should not be in dispute.
  ☑ Negative test
- Caller is not Inbox.
  ☑ Negative test

Function: `passAgentRoot()`

This function moves onto the next round of message.

Branches and code coverage (including function calls)

Intended branches

- Returns root not passed or pending if the agent root has not changed.
  ☑ Test coverage
- Returns root passed or not pending if the agent root optimistic period is over.
  ☑ Test coverage

Negative behavior

- Does not pass or is not pending if notary is in dispute.
  ☐ Negative test

Function call analysis

- `InterfaceLightManager(address(agentManager)).setAgentRoot(newRoot)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

4.5 Module: ExecutionHub.sol

Function: `execute(byte[] msgPayload, byte[32][] originProof, byte[32][] snapProof, uint256 stateIndex, uint64 gasLimit)`

Allows proven messages to be executed.

Inputs

- `msgPayload`
  - **Control**: Fully controlled by the caller.

  - **Constraints**: Should follow the message format.
  - **Impact**: The message to be executed.
- `originProof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid proof of inclusion of the message in the Origin Merkle tree.
  - **Impact**: The proof is used to reconstruct the Origin Merkle root.
- `snapProof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid proof of inclusion of the Origin state's left leaf into the Snapshot Merkle tree.
  - **Impact**: The proof is used to reconstruct the Snapshot Merkle root.
- `stateIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A in this function.
  - **Impact**: Index of Origin state in the Snapshot.
- `gasLimit`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be greater than the minimum threshold in `_executeBaseMessage`.
  - **Impact**: Gas limit for message execution.

Branches and code coverage (including function calls)

Intended branches

- Revert if message payload is not properly formatted (done in `castToMessage`).
  ☑ Test coverage
- Revert if snapshot root (reconstructed from message hash and proofs) is unknown (done in `_proveAttestation`).
  ☑ Test coverage
- Revert if snapshot root is known but was submitted by an inactive notary (done in `_proveAttestation`).
  ☑ Test coverage
- Revert if snapshot root is known but optimistic period for a message has not passed (done in `_proveAttestation`).
  ☑ Test coverage
- Revert if provided gas limit is lower than the one requested in the message (done in `_executeBaseMessage`).
  ☑ Test coverage

- Revert if recipient does not implement a `handle` method (refer to IMessageRecipient.sol; done in `_executeBaseMessage`).
  ☑ Test coverage
- Revert if recipient reverted upon receiving a message (done in `_executeBaseMessage`).
  ☑ Test coverage

Negative behavior

- Should not allow invalid proofs to be used to reconstruct the Origin Merkle root.
  ☑ Negative test
- Should not allow reusing the same parameters to execute the same message twice.
  ☑ Negative test

Function call analysis

- `IMessageRecipient(recipient).receiveBaseMessage{gas: gasLimit}({origin: header.origin(), nonce: header.nonce(), sender: baseMessage.sender(), proofMaturity: proofMaturity, version: request.version(), content: baseMessage.content().clone()})`
  - **What is controllable?** `baseMessage`.
  - **If return value controllable, how is it used and how can it go wrong?** No return, though it can revert.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts twice, the transaction should revert altogether, otherwise it should be fine.

Function: `_proveAttestation(Header header, byte[32] msgLeaf, byte[32][] originProof, byte[32][] snapProof, uint256 stateIndex)`

Attempts to prove the validity of the cross-chain message.

Inputs

- `header`
  - **Control**: Fully controlled by the caller (INTERNAL CALL).
  - **Constraints**: N/A in this function.
  - **Impact**: The header of the message to be executed.
- `msgLeaf`
  - **Control**: Fully controlled by the caller (INTERNAL CALL).
  - **Constraints**: N/A in this function.

  - **Impact**: The message leaf that was inserted in the Origin Merkle tree.
- `originProof`
  - **Control**: Fully controlled by the caller (INTERNAL CALL).
  - **Constraints**: Should be a valid proof of inclusion of the message in the Origin Merkle Tree.
  - **Impact**: The proof is used to reconstruct the Origin Merkle root.
- `snapProof`
  - **Control**: Fully controlled by the caller (INTERNAL CALL).
  - **Constraints**: Should be a valid proof of inclusion of the Origin state's left leaf into Snapshot Merkle tree.
  - **Impact**: The proof is used to reconstruct the Snapshot Merkle root.
- `stateIndex`
  - **Control**: Fully controlled by the caller (INTERNAL CALL).
  - **Constraints**: N/A in this function.
  - **Impact**: Index of Origin state in the Snapshot.

Branches and code coverage (including function calls)

Intended branches

- Revert if origin proof length exceeds Origin tree height.
  ☑ Test coverage
- Revert if snapshot proof length exceeds Snapshot tree height.
  ☑ Test coverage
- Revert if Snapshot Root has not been submitted.
  ☑ Test coverage
- Revert if notary who submitted the attestation is in dispute.
  ☑ Test coverage

Negative behavior

- Should not allow invalid proofs to be used to reconstruct the Origin Merkle root.
  ☑ Negative test
- Should not allow invalid proofs to be used to reconstruct the Snapshot Merkle root.
  ☑ Negative test
- Should not allow reusing the same parameters to execute the same message twice.
  ☑ Negative test

4.6 Module: GasOracle.sol

Function: `setGasData(uint32 domain, uint256 gasPrice, uint256 dataPrice, uint256 execBuffer, uint256 amortAttCost, uint256 etherPrice, uint256 markup)`

Sets gas data for variables that cannot be calculated on chain, `onlyOwner`.

Inputs

- `domain`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The domain (chain).
- `gasPrice`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Price of gas.
- `dataPrice`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Price of data.
- `execBuffer`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Execution buffer.
- `amortAttCost`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amortized cost of attestation (average cost of attestation).
- `etherPrice`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Price of Ether.
- `markup`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Markup.

Branches and code coverage (including function calls)

Intended branches

- Updates `gasData` if the new `GasData` is different.
  ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  ☑ Negative test

Function: `updateGasData(uint32 domain)`

Fetches local gas domain values and updates the gas data if they are new.

Inputs

- `domain`
  - **Control**: Full.
  - **Constraints**: Should only update if domain == local domain.
  - **Impact**: Domain.

Branches and code coverage (including function calls)

Intended branches

- Updates gas data if domain == local domain.
  ☑ Test coverage

Negative behavior

- Does not update gas data if domain != local domain.
  ☑ Negative test

4.7 Module: Inbox.sol

Function: `initialize(address agentManager_, address origin_, address destination_, address summit_)`

Initializes the contract.

Inputs

- `agentManager_`

  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the AgentManager contract.
- `origin_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the Origin contract.
- `destination_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the Destination contract.
- `summit_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the Summit contract.

Branches and code coverage (including function calls)

Intended branches

- Should call all underlying initializers (`StatementInbox`, `Ownable`).
  ☑ Test coverage
- Set the `summit` variable.
  ☑ Test coverage
- Set the owner to `msg.sender`.
  ☑ Test coverage
- Set the `agentManager` variable.
  ☑ Test coverage
- Set the `origin` variable.
  ☑ Test coverage
- Set the `destination` variable.
  ☑ Test coverage

Negative behavior

- Should not be callable multiple times.
  ☑ Negative test

Function: `passReceipt(uint32 attNotaryIndex, uint32 attNonce, uint256 paddedTips, byte[] rcptPayload)`

Passes the message execution receipt to Summit.

Inputs

- `attNotaryIndex`
  - **Control**: Fully controlled by the caller (Destination contract).
  - **Constraints**: N/A; assumed checked by Destination contract.
  - **Impact**: The index of the notary that signed the attestation.
- `attNonce`
  - **Control**: Fully controlled by the caller (Destination contract).
  - **Constraints**: N/A; assumed checked by Destination contract.
  - **Impact**: The nonce of the attestation.
- `paddedTips`
  - **Control**: Fully controlled by the caller (Destination contract).
  - **Constraints**: N/A; assumed checked by Destination contract.
  - **Impact**: The tips for the message execution.
- `rcptPayload`
  - **Control**: Fully controlled by the caller (Destination contract).
  - **Constraints**: N/A; assumed checked by Destination contract.
  - **Impact**: The payload of the message execution receipt.

Branches and code coverage (including function calls)

Intended branches

- Pass the message on to `summit` via calling `InterfaceSummit.acceptReceipt`.
  ☑ Test coverage

Negative behavior

- Should not allow anyone other than the Destination contract to call this function.
  ☑ Negative test

Function call analysis

- `summit.acceptReceipt(rcptNotaryIndex, attNotaryIndex, type(uint256).max, attNonce, paddedTips, rcptPayload)`
  - **What is controllable?** `attNotaryIndex`, `attNonce`, `paddedTips`, and `rcptPayload`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Means payload is not a receipt.

Function: `submitReceiptReport(byte[] rcptPayload, byte[] rcptSignature, byte[] rrSignature)`

Allows receiving a guard's receipt report along with the notary's signature for the reported receipt.

Inputs

- `rcptPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it fits the receipt format; will revert if it does not.
  - **Impact**: The payload of the receipt.
- `rcptSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is valid for the given receipt object.
  - **Impact**: The receipt signature.
- `rrSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is valid for the given receipt object.
  - **Impact**: The receipt report signature.

Branches and code coverage (including function calls)

Intended branches

- Verify that the receipt report is valid based on the signature.
  ☑ Test coverage
- Verify that the receipt is valid based on the signature.
  ☑ Test coverage
- Verify that the receipt report signer is a known guard.
  ☑ Test coverage
- Verify that the receipt signer is a known notary.
  ☑ Test coverage
- Should save the report.
  ☑ Test coverage

Negative behavior

- Should not allow invalid receipt formats.

  - ☑ Negative test
- Should not allow reports from inactive agents.
  - ☑ Negative test
- Should not allow either of the signers to be in dispute.
  - ☑ Negative test

Function call analysis

- `agentManager.openDispute(guardStatus.index, notaryStatus.index)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means that either are in dispute already.

Function: `submitReceipt(byte[] rcptPayload, byte[] rcptSignature, uint256 paddedTips, byte[32] headerHash, byte[32] bodyHash)`

Allows user to submit a receipt signed by a notary.

Inputs

- `rcptPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have a valid receipt format — otherwise, revert.
  - **Impact**: The receipt payload.
- `rcptSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the receipt object with it.
  - **Impact**: The signature for the receipt payload.
- `paddedTips`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the correct amount has been given.
  - **Impact**: The tips according to the receipt payload.
- `headerHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to retrieve the tips' receipt with it. Part of the Merkle proof.
  - **Impact**: The hash of the message header.
- `bodyHash`
  - **Control**: Fully controlled by the caller.

- **Constraints**: Should be able to retrieve the tips' receipt with it. Part of the message Merkle proof.
- **Impact**: The hash of the message body without the tips.

Branches and code coverage (including function calls)

Intended branches

- Should allow passing the message execution receipt to Summit.
  ☑ Test coverage
- Should save the used signature.
  ☑ Test coverage

Negative behavior

- Should not allow invalid receipt formats.
  ☑ Negative test
- Should not allow receipts from inactive or unknown notaries.
  ☑ Negative test
- Should not allow receipts from notaries that do not have the same domain as the receipt's destination domain.
  ☑ Negative test
- Should not allow less tips to be sent than necessary.
  ☑ Negative test
- Should not allow receipts from a notary that is in dispute.
  ☑ Negative test

Function call analysis

- `summit.acceptReceipt(rcptNotaryIndex, attNotaryIndex, sigIndex, attNonce, paddedTips, rcptPayload)`
  - **What is controllable?** `paddedTips` and `rcptPayload`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns `wasAccepted`. This, however, is currently never checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means that the payload is not a receipt; also it could mean that the notary is in dispute.

Function: `submitSnapshot(byte[] snapPayload, byte[] snapSignature)`

Accepts the snapshot signed by a guard or a notary and passes it to Summit.

Inputs

- `snapPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it fits the snapshot format; will revert if it does not.
  - **Impact**: The payload of the snapshot.
- `snapSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Verified when recovering the signed. Will revert if it does not pass verification.
  - **Impact**: The signature for the snapshot.

Branches and code coverage (including function calls)

Intended branches

- Accept guard snapshot if the snapshot is signed by a guard.
  ☑ Test coverage
- Accept notary snapshot if the snapshot is signed by a notary.
  ☑ Test coverage
- Accept the attestations if the snapshot is signed by a notary.
  ☑ Test coverage
- Pass the attestation to the Destination contract.
  ☑ Test coverage

Negative behavior

- Not allow invalid snapshots.
  ☑ Negative test
- Not allow snapshots from inactive agents.
  ☑ Negative test
- Not allow snapshots from agents in dispute.
  ☑ Negative test
- Not allow snapshots from agents with a nonce smaller than previously submitted.
  ☑ Negative test

Function call analysis

- `summit.acceptGuardSnapshot(status.index, sigIndex, snapPayload)`
  - **What is controllable?** `snapPayload`.

  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means that the payload is not a snapshot.
- `agentManager.agentRoot()`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Return not controllable; returns the root of the agent Merkle tree.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `summit.acceptNotarySnapshot(status.index, sigIndex, agentRoot, snapPayload)`
  - **What is controllable?** `snapPayload` and `agentRoot`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the attestation payload. Controlled by the fact that `summit` can only be set once, at initialization.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means that the payload is not a snapshot; also it could mean that the notary is in dispute.
- `destination.acceptAttestation(status.index, sigIndex, attPayload, agentRoot, snapGas)`
  - **What is controllable?** `attPayload` and `agentRoot`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns `wasAccepted`. This, however, is currently never checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means that the payload is not a snapshot; also it could mean that the notary is in dispute or that the snapshot's root has been previously submitted.

Function: `verifyAttestationReport(byte[] attPayload, byte[] arSignature)`

Verifies the guard's attestation report.

Inputs

- `attPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should fit the attestation format; will revert if it does not.
  - **Impact**: The payload of the attestation.
- `arSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked against the attestation payload; will revert if it does

not pass verification.
  - **Impact**: The signature for the attestation report.

Branches and code coverage (including function calls)

Intended branches

- Slash the signer if the attestation is valid.
  ☑ Test coverage
- Assure that the attestation is checked against the signature.
  ☑ Test coverage

Negative behavior

- Should not allow passing invalid payloads that have not been signed.
  ☑ Negative test

Function call analysis

- `summit.isValidAttesation(attPayload)`
  - **What is controllable?** `attPayload`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns `isValid`. Report is valid if the attestation is invalid.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `agentManager.slashAgent(status.domain, guard, msg.sender)`
  - **What is controllable?** `msg.sender` and partly the `guard` address and `status.domain`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if the domain is invalid or the caller is not this contract.

Function: `verifyAttestation(byte[] attPayload, byte[] attSignature)`

Verifies the notary's attestation.

Inputs

- `attPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should fit the attestation format; will revert if it does not.
  - **Impact**: The payload of the attestation.
- `attSignature`

- **Control**: Fully controlled by the caller.
- **Constraints**: Checked against the attestation payload; will revert if it does not pass verification.
- **Impact**: The signature for the attestation.

Branches and code coverage (including function calls)

Intended branches

- Slash the signer if the attestation is invalid.
  ☑ Test coverage

Negative behavior

- Should not allow passing invalid payloads that have not been signed.
  ☑ Negative test

Function call analysis

- `summit.isValidAttesation(attPayload)`
  - **What is controllable?** `attPayload`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns `isValid`, which tells whether the attestation is valid or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `agentManager.slashAgent(status.domain, notary, msg.sender)`
  - **What is controllable?** `msg.sender` and partly the `notary` address and `status.domain`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if the domain is invalid or the caller is not this contract.

4.8 Module: LightInbox.sol

Function: `initialize(address agentManager_, address origin_, address destination_)`

Initializes the contract.

Inputs

- `agentManager_`

  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the AgentManager contract.
- `origin_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the Origin contract.
- `destination_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address of the Destination contract.

Branches and code coverage (including function calls)

Intended branches

- Should call all underlying initializers (`StatementInbox`, `Ownable`).
  ☑ Test coverage
- Set the owner to `msg.sender`.
  ☑ Test coverage
- Set the `agentManager` variable.
  ☑ Test coverage
- Set the `origin` variable.
  ☑ Test coverage
- Set the `destination` variable.
  ☑ Test coverage

Negative behavior

- Should not be callable multiple times.
  ☑ Negative test

Function: `submitAttestationReport(byte[] attPayload, byte[] arSignature, byte[] attSignature)`

Allow guard to submit a report for a presumably malicious attestation.

Inputs

- `attPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid attestation format — revert otherwise.

  - **Impact**: The reported attestation.
- `arSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Validated against the attestation itself.
  - **Impact**: The report signature.
- `attSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Validated against the attestation itself.
  - **Impact**: The attestation signature.

Branches and code coverage (including function calls)

Intended branches

- Should open a dispute between the reporting guardian and the reported notary.
  ☑ Test coverage
- Should save the report.
  ☑ Test coverage

Negative behavior

- Should not allow an invalid attestation format.
  ☑ Negative test
- Should not allow invalid attestation or report signatures.
  ☑ Negative test
- Should not allow signatures from notaries or guards that are either inactive or unknown.
  ☑ Negative test
- Should not allow a notary that does not belong to the current chain.
  ☑ Negative test
- Should not allow either actors to be in dispute.
  ☑ Negative test

Function call analysis

- `agentManager.oopen Dispute(guardStatus.index, notaryStatus.index)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means either the notary/guard are in dispute.

Function: `submitAttestation(byte[] attPayload, byte[] attSignature, byte[32] agentRoot_, uint256[] snapGas_)`

Allows for accepting an attestation that has been previously signed by a notary, then passing it to the destination.

Inputs

- `attPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have the attestation format.
  - **Impact**: The attestation data.
- `attSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Verified against the given attestation.
  - **Impact**: The notary's signature for the attestation.
- `agentRoot_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: To match with the attestation hash.
  - **Impact**: The Merkle root for the given attestation
- `snapGas_`
  - **Control**: Fully controlled by caller.
  - **Constraints**: Used in forwarding the message to calculate the gas data for each chain.
  - **Impact**: Gas data for snapshots.

Branches and code coverage (including function calls)

Intended branches

- Pass the attestation on to the Destination contract.
  ☑ Test coverage
- Save the attestation signature locally.
  ☑ Test coverage

Negative behavior

- Should not allow invalid attestation payload formats.
  ☑ Negative test
- Should not allow an unknown or inactive notary's attestation.
  ☑ Negative test
- Should not allow a notary that is not active on the current chain.

☑ Negative test

- Should not allow an invalid `agentRoot` when recovering the attestation data hash.
  ☑ Negative test
- Should not allow a disputed notary to submit attestations.
  ☑ Negative test

Function call analysis

- `destination.acceptAttestation(status.index, sigIndex, attPayload, agentRoot, snapGas)`
  - **What is controllable?** `attPayload` and `agentRoot`.
  - **If return value controllable, how is it used and how can it go wrong?** Returns `wasAccepted`. This, however, is currently never checked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means that the payload is not a snapshot; also it could mean that the notary is in dispute or that the snapshot's root has been previously submitted.

4.9 Module: LightManager.sol

Function: `remoteWithdrawTips(uint32 msgOrigin, uint256 proofMaturity, address recipient, uint256 amount)`

Withdraw tips — just an interface to withdrawing tips. Checks happen here.

Inputs

- `msgOrigin`
  - **Control**: Full.
  - **Constraints**: Must be `SynapseDomain`.
  - **Impact**: The origin domain (chain) of the message.
- `proofMaturity`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The Merkle proof maturity (this is all passed by the destination, so it should be trusted).
- `recipient`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Recipient.
- `amount`

  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Tip amount to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Calls origin with `withdrawTips`.
  ☑ Test coverage

Negative behavior

- Caller, not destination.
  ☑ Negative test
- Source domain, not Synapse.
  ☑ Negative test
- Optimistic period not over.
  ☑ Negative test

Function call analysis

- `InterfaceOrigin(origin).withdrawTips(recipient, amount)`
  - **What is controllable?** Full.
  - **If return value controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Discarded.

Function: `updateAgentStatus(address agent, AgentStatus status, byte[32][] proof)`

Updates the agent root (the Merkle tree containing all agent statuses).

Inputs

- `agent`
  - **Control**: Full.
  - **Constraints**: Agent `==` `storedAgent`.
  - **Impact**: The agent in question.
- `status`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The new status.

- `proof`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The Merkle tree proof.

Branches and code coverage (including function calls)

Intended branches

- Agent status should be updated if the proofs are correct (active/slashed).
  ☑ Test coverage

Negative behavior

- Reverts if agent root does not match.
  ☑ Negative test
- Will revert if agent slashed earlier.
  ☐ Negative test

4.10 Module: MultiCallable.sol

Function: `multicall(Call[] calls)`

Allows a user to make multiple calls to this contract in a single transaction.

Inputs

- `calls`
  - **Control**: Fully controlled by caller.
  - **Constraints**: None; can be empty.
  - **Impact**: Each call will be made to this contract, in order. If any call fails, the entire transaction will revert.

Branches and code coverage (including function calls)

Intended branches

- Should revert if any call fails and `allowFailure` is false.
  ☑ Test coverage
- Call the contract with the given `callData` for each call in `calls`.
  ☑ Test coverage
- Assure that all calls are made in order.
  ☑ Test coverage

Negative behavior

- Should NOT be payable. That is because, under multiple calls, the value of `msg.value` is maintained.
  ☑ Negative test

4.11 Module: Origin.sol

Function: `sendBaseMessage(uint32 destination, byte[32] recipient, uint32 optimisticPeriod, uint256 paddedRequest, byte[] content)`

The entry point of the CrossChain bridge

Inputs

- `destination`
  - **Control**: Full.
  - **Constraints**: Must be a chain ID.
  - **Impact**: Chain ID.
- `recipient`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Recipient.
- `optimisticPeriod`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Period for this message to go through the system and the various fraud checks.
- `paddedRequest`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The padded request.
- `content`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The message.

Branches and code coverage (including function calls)

Intended branches

- End-to-end test for cross-chain messaging.
  ☑ Test coverage

Negative behavior

- Should not accept tips below minimum.
  ☑ Negative test

Function: `sendManagerMessage(uint32 destination, uint32 optimisticPeriod, byte[] payload)`

Sends an agent manager message.

Inputs

- `destination`
  - **Control**: Full.
  - **Constraints**: Valid chain ID.
  - **Impact**: Chain ID.
- `optimisticPeriod`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The period to wait for the message to go through the system.
- `payload`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The message in bytes.

Branches and code coverage (including function calls)

Intended branches

- Transfers a cross-chain manager message.
  ☐ Test coverage

Negative behavior

- Can be called by roles other than `agentManager`.
  ☐ Negative test

4.12 Module: SnapshotHub.sol

Function: `_acceptGuardSnapshot(Snapshot snapshot, uint32 guardIndex, uint256 sigIndex)`

Accepts a snapshot signed by a guard.

Branches and code coverage (including function calls)

Intended branches

- Save the signed state.
  ☑ Test coverage
- Assumes all checks have been done by this point by the function calling this one.
  ☑ Test coverage
- Push the newly added snapshot to the `_guardSnapshots` array.
  ☑ Test coverage

Negative behavior

- Should not allow a state pointer to be zero.
  ☑ Negative test
- Should not allow the snapshot's state nonce to be less than the latest state nonce (checked in `saveState`).
  ☑ Negative test

Function: `_acceptNotarySnapshot(Snapshot snapshot, bytes32 agentRoot, uint32 notaryIndex, uint256 sigIndex)`

Accepts a snapshot signed by a notary.

Branches and code coverage (including function calls)

Intended branches

- Save the signed state.
  ☑ Test coverage
- Assumes all checks have been done by this point by the function calling this one.
  ☑ Test coverage
- Push the newly added snapshot to the `_notarySnapshots` array.
  ☑ Test coverage
- Return the resulting attestation.
  ☑ Test coverage
- Ensure that `attestations.length == notarySnapshots.length` after the function is called.
  ☐ Test coverage

Negative behavior

- Should not allow the snapshot's state nonce to be less than the latest state nonce (checked in `saveState`).
  ☑ Negative test
- Should not allow using states that have NOT been previously submitted by any of the guards.
  ☑ Negative test

4.13 Module: Snapshot.sol

Function: `calculateRoot(Snapshot snapshot)`

Calculates the root of a snapshot (Merkle tree root).

Inputs

- `snapshot`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The snapshot.

Branches and code coverage (including function calls)

Intended branches

- Generate the expected root of the Merkle tree according to the hashes.
  ☑ Test coverage

Function: `proofSnapRoot(bytes32 originRoot, uint32 domain, bytes32[] memory snapProof, uint256 stateIndex)`

Prove a state in a snapshot is valid for a certain root, with the relevant proofs.

Inputs

- `originRoot`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The root to check against.
- `domain`
  - **Control**: Full.

  - **Constraints**: None.
  - **Impact**: The domain used in hashing of leaves.
- `snapProof`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The proofs for the Merkle tree.
- `stateIndex`
  - **Control**: Full.
  - **Constraints**: `< states.len`.
  - **Impact**: The state in the snapshot, which is verified as being in the Merkle tree.

Branches and code coverage (including function calls)

Intended branches

- proves that the merkle tree roots are equal
  ☑ Test coverage

Negative behaviour

- Should not allow `stateIndex` to be larger than the number of states.
  ☐ Negative test

4.14 Module: StatementInbox.sol

Function: `submitStateReportWithAttestation(uint256 stateIndex, byte[] srSignature, byte[] snapPayload, byte[] attPayload, byte[] attSignature)`

Allows submitting a state report with an attestation.

Inputs

- `stateIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid state index, within bounds.
  - **Impact**: The index of the state in the snapshot.
- `srSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the snapshot with it.

    - **Impact**: The signature for the guard that submitted the report.
- `snapPayload`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Should have valid snapshot format.
    - **Impact**: The snapshot that is reported.
- `attPayload`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Should have valid attestation format.
    - **Impact**: The attestation that is reported.
- `attSignature`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Should be able to verify the attestation with it.
    - **Impact**: The signature for the notary that submitted the attestation.

Branches and code coverage (including function calls)

Intended branches

- Open a dispute between the guard and the notary, should the report be valid.
  ☑ Test coverage

Negative behavior

- Should not allow invalid snapshot payload.
  ☑ Negative test
- Should not allow invalid attestation payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular snapshot.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular attestation.
  ☑ Negative test
- Should not allow reporting a snapshot from an unknown guard.
  ☑ Negative test
- Should not allow reporting a snapshot from an inactive guard.
  ☑ Negative test
- Should not allow reporting a snapshot of an unknown notary.
  ☑ Negative test
- Should not allow reporting a snapshot of an inactive notary.
  ☑ Negative test

- Should revert if the state index is out of range.
  ☑ Negative test
- Revert if the attestation root is not equal to the Merkle root derived from the state and snapshot proof.
  ☑ Negative test
- Should not allow using a snap payload that cannot be verified by both the `srSignature` and `snapSignature`.
  ☑ Negative test

Function call analysis

- `agentManager.openDispute(guardStatus.index, notaryStatus.index)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means either actor is already in dispute.

Function: `submitStateReportWithSnapshotProof(uint256 stateIndex, byte[] statePayload, byte[] srSignature, byte[32][] snapProof, byte[] attPayload, byte[] attSignature)`

Allows submitting a state report with a snapshot proof.

Inputs

- `stateIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid state index, within bounds.
  - **Impact**: The index of the state in the snapshot.
- `statePayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid state format.
  - **Impact**: The state that is reported.
- `srSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the state with it.
  - **Impact**: The signature for the guard that submitted the report.
- `snapProof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the state with it.
  - **Impact**: The snapshot proof that is reported.

- `attPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid attestation format.
  - **Impact**: The attestation that is reported.
- `attSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the attestation with it.
  - **Impact**: The signature for the notary that submitted the attestation.

Branches and code coverage (including function calls)

Intended branches

- Open a dispute between the guard and the notary, should the report be valid.
  ☑ Test coverage

Negative behavior

- Should not allow invalid state payload.
  ☑ Negative test
- Should not allow invalid attestation payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular state.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular snapshot.
  ☑ Negative test
- Should not allow reporting a state from an unknown guard.
  ☑ Negative test
- Should not allow reporting a state from an inactive guard.
  ☑ Negative test
- Should not allow reporting a state of an unknown notary.
  ☑ Negative test
- Should not allow reporting a state of an inactive notary.
  ☑ Negative test
- Should revert if the state index is out of range.
  ☑ Negative test
- Revert if the attestation root is not equal to the Merkle root derived from the state and snapshot proof.
  ☑ Negative test

- Should not allow using a snap payload that cannot be verified by both the `srSignature` and `snapSignature`.
  ☑ Negative test
- Revert if snapshot proof length exceeds snapshot tree height.
  ☑ Negative test
- Revert if snapshot proof first element does not match the state metadata.
  ☑ Negative test

Function call analysis

- `agentManager.openDispute(guardStatus.index, notaryStatus.index)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means either actor is already in dispute.

Function: `submitStateReportWithSnapshot(uint256 stateIndex, byte[] srSignature, byte[] snapPayload, byte[] snapSignature)`

Allows submitting a state report with a snapshot.

Inputs

- `stateIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid state index, within bounds.
  - **Impact**: The index of the state in the snapshot.
- `srSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the snapshot with it.
  - **Impact**: The signature for the guard that submitted the report.
- `snapPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have a valid snapshot format.
  - **Impact**: The snapshot that is reported.
- `snapSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the snapshot with it.
  - **Impact**: The signature for the guard that submitted the snapshot.

Branches and code coverage (including function calls)

Intended branches

- Open a dispute between the guard and the notary, should the report be valid.
  ☑ Test coverage

Negative behavior

- Should not allow invalid snapshot payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular snapshot.
  ☑ Negative test
- Should not allow reporting a snapshot from an unknown guard.
  ☑ Negative test
- Should not allow reporting a snapshot from an inactive guard.
  ☑ Negative test
- Should not allow reporting a snapshot of an unknown notary.
  ☑ Negative test
- Should not allow reporting a snapshot of an inactive notary.
  ☑ Negative test
- Should revert if the state index is out of range.
  ☑ Negative test
- Should not allow using a snap payload that cannot be verified by both the `srSignature and snapSignature`.
  ☑ Negative test

Function call analysis

- `agentManager.openDispute(guardStatus.index, notaryStatus.index)`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means either actor is already in dispute.

Function: `verifyReceiptReport(byte[] rcptPayload, byte[] rrSignature)`

Used to report a receipt sent by a guard.

Inputs

- `rcptPayload`

  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid receipt format.
  - **Impact**: The receipt that is reported.
- `rrSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to validate the receipt with it.
  - **Impact**: The reported receipt's signature.

Branches and code coverage (including function calls)

Intended branches

- Should do nothing if report is valid (i.e., reported receipt is invalid).
  ☑ Test coverage
- Should slash the guard, should the report be invalid (i.e., reported receipt is actually valid).
  ☑ Test coverage

Negative behavior

- Should not allow an invalid receipt payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular receipt.
  ☑ Negative test
- Should not allow reporting a receipt from an unknown guard.
  ☑ Negative test
- Should not allow reporting a receipt from an inactive guard.
  ☑ Negative test
- Should not slash the guard if the reported receipt is valid.
  ☑ Negative test

Function call analysis

- `origin.isValidReceipt(rcptPayload)`
  - **What is controllable?** `rcptPayload`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; it returns whether the receipt is valid or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means receipt format is invalid; it should not happen by this point.
- `agentManager.slashAgent(status.domain, guard, msg.sender)`
  - **What is controllable?** N/A — `msg.sender` to some extent.

- **If return value controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters, or does other unusual control flow?** Means the agent has been slashed already or that its domain does not match the status's domain.

Function: `verifyReceipt(byte[] rcptPayload, byte[] rcptSignature)`

Allows verifying a receipt.

Inputs

- `rcptPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid receipt format.
  - **Impact**: The receipt that is verified.
- `rcptSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the receipt with it.
  - **Impact**: The signature for the notary that submitted the receipt.

Branches and code coverage (including function calls)

Intended branches

- Should either slash the notary, should the receipt be invalid, or do nothing, should the receipt be valid.
  ☑ Test coverage

Negative behavior

- Should not allow invalid receipt payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular receipt.
  ☑ Negative test
- Should not allow reporting a receipt from an unknown notary.
  ☑ Negative test
- Should not allow reporting a receipt from an inactive notary.
  ☑ Negative test
- Should not slash the notary if the reported receipt is valid.
  ☑ Negative test

Function call analysis

- `agentManager.slashAgent(status.domain, notary, msg.sender)`
  - **What is controllable?** `msg.sender`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means the agent has been slashed already or that its domain does not match the status's domain.

Function: `verifyStateReport(byte[] statePayload, byte[] srSignature)`

Verifies a guard's state report.

Inputs

- `statePayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have a valid state format.
  - **Impact**: The state that is reported.
- `srSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the state with it.
  - **Impact**: The signature for the guard that submitted the state.

Branches and code coverage (including function calls)

Intended branches

- Should do nothing if report is valid (i.e., reported state is invalid).
  ☑ Test coverage
- Should slash the guard, should the report be invalid (i.e., reported state is actually valid).
  ☑ Test coverage

Negative behavior

- Should not allow an invalid state payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular state.
  ☑ Negative test
- Should not allow reporting a state from an unknown guard.
  ☑ Negative test

- Should not allow reporting a state from an inactive guard.
  ☑ Negative test
- Should not slash the guard if the reported state is valid.
  ☑ Negative test

Function call analysis

- `origin.isValidState(statePayload)`
  - **What is controllable?** `statePayload`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; it returns whether the state is valid or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means state format is invalid; it should not happen by this point.
- `agentManager.slashAgent(status.domain, guard, msg.sender)`
  - **What is controllable?** N/A — `msg.sender` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means the agent has been slashed already or that its domain does not match the status's domain.

Function: `verifyStateWithAttestation(uint256 stateIndex, byte[] snapPayload, byte[] attPayload, byte[] attSignature)`

Allows verifying a state with an attestation.

Inputs

- `stateIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid state index, within bounds.
  - **Impact**: The index of the state in the snapshot.
- `snapPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid snapshot format.
  - **Impact**: The snapshot payload.
- `attPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid attestation format.
  - **Impact**: The attestation with which the state is reported.
- `attSignature`
  - **Control**: Fully controlled by the caller.

- **Constraints**: Should be able to verify the attestation with it.
- **Impact**: The signature for the notary that submitted the attestation.

Branches and code coverage (including function calls)

Intended branches

- Should either slash the notary, should the state be invalid, or do nothing, should the state be valid.
  ☑ Test coverage

Negative behavior

- Should not allow invalid snapshot payload.
  ☑ Negative test
- Should not allow invalid attestation payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular snapshot.
  ☑ Negative test
- Should not allow reporting a snapshot from an unknown notary.
  ☑ Negative test
- Should not allow reporting a snapshot from an inactive notary.
  ☑ Negative test
- Should revert if the state index is out of range.
  ☑ Negative test
- Revert if the attestation root is not equal to the Merkle root derived from the state and snapshot proof.
  ☑ Negative test
- Should not allow using a snap payload that cannot be verified by both the `srSignature` and `snapSignature`.
  ☑ Negative test

Function call analysis

- `origin.isValidState(statePayload)`
  - **What is controllable?** `statePayload`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; it returns whether the state is valid or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means state format is invalid; it should not happen by this point.
- `agentManager.slashAgent(status.domain, guard, msg.sender)`

  - **What is controllable?** N/A — `msg.sender` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means the agent has been slashed already or that its domain does not match the status's domain.

Function: `verifyStateWithSnapshotProof(uint256 stateIndex, byte[] statePayload, byte[32][] snapProof, byte[] attPayload, byte[] attSignature)`

Allows verifying a state with a snapshot proof.

Inputs

- `stateIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be a valid state index, within bounds.
  - **Impact**: The index of the state in the snapshot.
- `statePayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid state format.
  - **Impact**: The state that is reported.
- `snapProof`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the state with it.
  - **Impact**: The snapshot proof that is reported.
- `attPayload`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should have valid attestation format.
  - **Impact**: The attestation that is reported.
- `attSignature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be able to verify the attestation with it.
  - **Impact**: The signature for the notary that submitted the attestation.

Branches and code coverage (including function calls)

Intended branches

- Should either slash the notary, should the state be invalid, or do nothing, should the state be valid.
  - ☑ Test coverage

Negative behavior

- Should not allow invalid state payload.
  ☑ Negative test
- Should not allow invalid attestation payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular state.
  ☑ Negative test
- Should not allow reporting a state from an unknown notary.
  ☑ Negative test
- Should not allow reporting a state from an inactive notary.
  ☑ Negative test
- Should revert if the state index is out of range.
  ☑ Negative test
- Revert if the attestation root is not equal to the Merkle root derived from the state and snapshot proof.
  ☑ Negative test
- Should not allow using a snap payload that cannot be verified by both the `srSignature` and `snapSignature`.
  ☑ Negative test
- Revert if snapshot proof length exceeds snapshot tree height.
  ☑ Negative test
- Revert if snapshot proof first element does not match the state metadata.
  ☑ Negative test

Function call analysis

- `origin.isValidState(statePayload)`
  - **What is controllable?** `statePayload`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; it returns whether the state is valid or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means state format is invalid; it should not happen by this point.
- `agentManager.slashAgent(status.domain, guard, msg.sender)`
  - **What is controllable?** N/A — `msg.sender` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means the agent has been slashed already or that its domain does not match the status's domain.

Function: `verifyStateWithSnapshot(uint256 stateIndex, byte[] snapPayload, byte[] snapSignature)`

Allows verifying a state with a snapshot.

Inputs

- `stateIndex`
  - **Control**: Fully controlled by caller.
  - **Constraints**: Should be a valid state index, within bounds.
  - **Impact**: The index of the state in the snapshot.
- `snapPayload`
  - **Control**: Fully controlled by caller.
  - **Constraints**: Should have valid snapshot format.
  - **Impact**: The snapshot that is reported.
- `snapSignature`
  - **Control**: Fully controlled by caller.
  - **Constraints**: Should be able to verify the snapshot with it.
  - **Impact**: The signature for the guard that submitted the snapshot.

Branches and code coverage (including function calls)

Intended branches

- Should either slash the guard, should the state be invalid, or do nothing, should the state be valid.
  ☑ Test coverage

Negative behavior

- Should not allow invalid snapshot payload.
  ☑ Negative test
- Should not allow using a signature that has not previously signed the particular snapshot.
  ☑ Negative test
- Should not allow reporting a snapshot from an unknown guard.
  ☑ Negative test
- Should not allow reporting a snapshot from an inactive guard.
  ☑ Negative test
- Should revert if the state index is out of range.
  ☑ Negative test
- Should not allow using a snap payload that cannot be verified by both the `srSi`

gnature and snapSignature.
  ☑ Negative test

Function call analysis

- origin.isValidState(statePayload)
  - **What is controllable?** statePayload.
  - **If return value controllable, how is it used and how can it go wrong?** N/A; it returns whether the state is valid or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means state format is invalid; it should not happen by this point.
- agentManager.slashAgent(status.domain, guard, msg.sender)
  - **What is controllable?** N/A — msg.sender to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means the agent has been slashed already or that its domain does not match the status's domain.

4.15 Module: Summit.sol

Function: acceptGuardSnapshot(uint32 guardIndex, uint256 sigIndex, byte[] snapPayload)

Accepts a guard snapshot.

Inputs

- guardIndex
  - **Control**: Full.
  - **Constraints**: Verified to be a guard in the inbox.
  - **Impact**: The index of the guard.
- sigIndex
  - **Control**: Full.
  - **Constraints**: Index of the sig in the list, guaranteed by Inbox.
  - **Impact**: The index of the sig in the list.
- snapPayload
  - **Control**: Full.
  - **Constraints**: From the inbox, controlled to be from a guard.
  - **Impact**: The snapshot in bytes format.

Branches and code coverage (including function calls)

Intended branches

- Explicitly tested on its own to accept guard snapshots.
  - ☐ Test coverage
- Saves the guard snapshot into the relevant state variables.
  - ☐ Test coverage

Negative behavior

- Cannot be called by anyone but the inbox.
  - ☑ Negative test
- Reverts when not provided a snapshot memview.
  - ☐ Negative test

Function: `acceptNotarySnapshot(uint32 notaryIndex, uint256 sigIndex, bytes32 agentRoot, byte[] snapPayload)`

Accepts a notary snapshot.

Inputs

- `notaryIndex`
  - **Control**: Full.
  - **Constraints**: Comes from inbox verified to not be signature.
  - **Impact**: Index of notary in list/mapping.
- `sigIndex`
  - **Control**: Full.
  - **Constraints**: Guaranteed.
  - **Impact**: Index of the sig in the sig list.
- `agentRoot`
  - **Control**: Full.
  - **Constraints**: Only Inbox, guaranteed to be the root of the tree.
  - **Impact**: Root of the agent Merkle tree.
- `snapPayload`
  - **Control**: Full.
  - **Constraints**: The snap payload made sure to be verified by the inbox.
  - **Impact**: Snap payload in bytes format.

Branches and code coverage (including function calls)

Intended branches

- Snapshot is accepted and relevant states are changed
  ☑ Test coverage

Negative behavior

- agents in dispute cannot accept snapshot
  ☑ Negative test

Function: `acceptReceipt(uint32 rcptNotaryIndex, uint32 attNotaryIndex, uint256 sigIndex, uint32 attNonce, uint256 paddedTips, byte[] rcptPayload)`

Accepts and confirms a receipt.

Inputs

- `rcptNotaryIndex`
  - **Control**: Full.
  - **Constraints**: Only Inbox.
  - **Impact**: The receipt notaries' index.
- `attNotaryIndex`
  - **Control**: Full.
  - **Constraints**: Only Inbox.
  - **Impact**: Attestation notary index.
- `sigIndex`
  - **Control**: Full.
  - **Constraints**: Only Inbox.
  - **Impact**: Unused.
- `attNonce`
  - **Control**: Full.
  - **Constraints**: Only Inbox.
  - **Impact**: Attestation nonce.
- `paddedTips`
  - **Control**: Full.
  - **Constraints**: Only Inbox.
  - **Impact**: The tips in decoded format.
- `rcptPayload`
  - **Control**: Full.
  - **Constraints**: Only Inbox.
  - **Impact**: Receipt byte message in untyped form.

Branches and code coverage (including function calls)

Intended branches

- Accepts the receipt and puts them into the `receiptQueue`.
  - ☐ Test coverage
- Modifies the relevant states in the contract.
  - ☐ Test coverage

Negative behavior

- Do not allow a `rcptNotary` who is in dispute to accept/submit a receipt.
  - ☐ Negative test
- Does not alter state if the status is still the same.
  - ☐ Negative test

Function: `distributeTips()`

Distributes the tips (does not send them, just changes the state to allocate tips).

Branches and code coverage (including function calls)

Intended branches

- Tips are distributed.
  - ☑ Test coverage
- Agents that have not passed the optimistic period are moved to the back of the queue.
  - ☐ Test coverage

Negative behavior

- Does not distribute tips if the queue is empty.
  - ☑ Negative test
- Does not distribute tips if the optimistic bonding period has not passed.
  - ☑ Negative test
- Does not distribute tips if the notary is in a dispute.
  - ☑ Negative test
- Does not distribute tips if the notary is fraudulent.
  - ☑ Negative test
- Does not distribute tips if the notary is fraudulent.
  - ☑ Negative test

4.16 Module: Tips.sol

Function: `matchValue(Tips tips, uint256 newValue)`

Matches the `msg.value`.

Inputs

- `tip`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The tip.
- `newValue`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The `msg.value`.

Branches and code coverage (including function calls)

Intended branches

- Matches the values and produces a new tip.
  ☑ Test coverage

Negative behavior

- Reverts if `newValue` is too low.
  ☑ Negative test

5 Audit Results

At the time of our audit, the audited code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Synapse Sanguine contracts, we discovered three findings. No critical issues were found. One was of medium impact, one was of low impact, and the remaining finding was informational in nature. Synapse acknowledged all findings and implemented fixes.

5.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.