CODESPECT

AlphaHype contracts

SECURITY ASSESSMENT REPORT

October 30, 2025

Prepared for:

α ticks

Contents

1 About CODESPECT

CODESPECT is a specialized smart contract security firm dedicated to ensure the safety, reliability, and success of blockchain projects. Our services include comprehensive smart contract audits, secure design and architecture consultancy, and smart contract development across leading blockchain platforms such as Ethereum (Solidity), Starknet (Cairo), and Solana (Rust).

At CODESPECT, we are committed to build secure, resilient blockchain infrastructures. We provide strategic guidance and technical expertise, working closely with our partners from concept development through deployment. Our team consists of blockchain security experts and seasoned engineers who apply the latest auditing and security methodologies to help prevent exploits and vulnerabilities in your smart contracts.

Smart Contract Auditing: Security is at the core of everything we do at CODESPECT. Our auditors conduct thorough security assessments of smart contracts written in Solidity, Cairo, and Rust, ensuring that they function as intended without vulnerabilities. We specialize in providing tailored security solutions for projects on EVM-compatible chains and Starknet. Our audit process is highly collaborative, keeping clients involved every step of the way to ensure transparency and security. Our team is also dedicated to cutting-edge research, ensuring that we stay ahead of emerging threats.

Secure Design & Architecture Consultancy: At CODESPECT, we believe that secure development begins at the design phase. Our consultancy services offer deep insights into secure smart contract architecture and blockchain system design, helping you build robust, secure, and scalable decentralized applications. Whether you're working with Ethereum, Starknet, or other blockchain platforms, our team helps you navigate the complexity of blockchain development with confidence.

Tailored Cybersecurity Solutions: CODESPECT offers specialized cybersecurity solutions designed to minimize risks associated with traditional attack vectors, such as phishing, social engineering, and Web2 vulnerabilities. Our solutions are crafted to address the unique security needs of blockchain-based applications, reducing exposure to attacks and ensuring that all aspects of the system are fortified.

With a focus on the intersection of security and innovation, CODESPECT strives to be a trusted partner for blockchain projects at every stage of development and for each aspect of security.

2 Disclaimer

Limitations of this Audit: This report is based solely on the materials and documentation provided to CODESPECT for the specific purpose of conducting the security review outlined in the Summary of Audit and Files. The findings presented in this report may not be comprehensive and may not identify all possible vulnerabilities. CODESPECT provides this review and report on an "as-is" and "as-available" basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, is entirely at your own risk.

Inherent Risks of Blockchain Technology: Blockchain technology is still evolving and is inherently subject to unknown risks and vulnerabilities. This review focuses exclusively on the smart contract code provided and does not cover the compiler layer, underlying programming language elements beyond the reviewed code, or any other potential security risks that may exist outside of the code itself.

Purpose and Reliance of this Report: This report should not be viewed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. Third parties should not rely on this report for any purpose, including making decisions related to investments or purchases.

Liability Disclaimer: To the maximum extent permitted by law, CODESPECT disclaims all liability for the contents of this report and any related services or products that arise from your use of it. This includes but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services: CODESPECT does not warrant, endorse, or assume responsibility for any third-party products or services mentioned in this report, including any open-source or third-party software, code, libraries, materials, or information that may be linked to, referenced by, or accessible through this report. CODESPECT is not responsible for monitoring any transactions between you and third-party providers. We strongly recommend conducting thorough due diligence and exercising caution when engaging with third-party products or services, just as you would for any other product or service transaction.

Further Recommendations: We advise clients to schedule a re-audit after any significant changes to the codebase to ensure ongoing security and reduce the risk of newly introduced vulnerabilities. Additionally, we recommend implementing a bug bounty program to incentivize external developers and security researchers to identify and disclose potential vulnerabilities safely and responsibly.

Disclaimer of Advice: FOR AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, AND ANY ASSOCIATED SERVICES OR MATERIALS SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Risk Classification

Severity Level	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Table 1: Risk Classification Matrix based on Likelihood and Impact

3.1 Impact

– **High** - Results in a substantial loss of assets (more than 10%) within the protocol or causes significant disruption to the majority of users.

– **Medium** - Losses affect less than 10% globally or impact only a portion of users, but are still considered unacceptable.

– **Low** - Losses may be inconvenient but are manageable, typically involving issues like griefing attacks that can be easily resolved or minor inefficiencies such as gas costs.

3.2 Likelihood

– **High** - Very likely to occur, either easy to exploit or difficult but highly incentivized.

– **Medium** - Likely only under certain conditions or moderately incentivized.

– **Low** - Unlikely unless specific conditions are met, or there is little-to-no incentive for exploitation.

3.3 Action Required for Severity Levels

– **Critical** - Must be addressed immediately if already deployed.

– **High** - Must be resolved before deployment (or urgently if already deployed).

– **Medium** - It is recommended to fix.

– **Low** - Can be fixed if desired but is not crucial.

In addition to High, Medium, and Low severity levels, CODESPECT utilizes two other categories for findings: **Informational** and **Best Practices**.

a) **Informational** findings do not pose a direct security risk but provide useful information the audit team wants to communicate formally.

b) **Best Practices** findings indicate that certain portions of the code deviate from established smart contract development standards.

4 Executive Summary

This document presents the security assessment conducted by CODESPECT for the smart contracts of αHYPE. αHYPE is a liquid staking token that you receive when staking HYPE with Alphaticks.

This audit focuses on the third version of the AlphaManager contract, which manages deposits and withdrawals and serves as the representation of the HYPE token.

The audit was performed using:

a) Manual analysis of the codebase.

b) Dynamic analysis of smart contracts, execution testing.

CODESPECT found four points of attention, one classified as `Medium`, two classified as `Informational`, and one classified as `Best Practices`. All of the issues are summarised in Table 2.

Organisation of the document is as follows:

- **Section 5** summarizes the audit.
- **Section 6** describes the functionality of the code in scope.
- **Section 7** presents the issues.
- **Section 8** discusses the documentation provided by the client for this audit.
- **Section 9** presents the compilation and tests.

Issues found:

Severity	Unresolved	Fixed	Acknowledged
Medium	0	1	0
Informational	0	0	2
Best Practices	0	0	1
Total	**0**	**1**	**3**

Table 2: Summary of Unresolved, Fixed, and Acknowledged Issues

5 Audit Summary

Audit Type	Security Review
Project Name	AlphaHYPE
Type of Project	Liquid Staking Token
Duration of Engagement	3 Days
Duration of Fix Review Phase	1 Day
Draft Report	October 27, 2025
Final Report	October 30, 2025
Repository	hfun-ahype
Commit (Audit)	5b8381ff277085280ed19ab069668fc8f2a47c8e
Commit (Final)	cac8c9e457602a9778506d07c09a45894866c00e
Documentation Assessment	Medium
Test Suite Assessment	Medium
Auditors	talfao, suspiciousbandicoot

Table 3: Summary of the Audit

5.1 Scope - Audited Files

	Contract	**LoC**
1	AlphaHYPEManager03.sol	278
2	libraries/HcorePrecompiles.sol	270
3	libraries/Math.sol	18
4	libraries/SafeMath.sol	12
	Total	**578**

5.2 Findings Overview

	Finding	Severity	Update
1	Stale precompile data leads to incorrect pricing and loss for withdrawers	Medium	Fixed
2	HyperCore enforces 5-request withdrawal cap	Info	Acknowledged
3	Undelegation may silently fail due to validator lock	Info	Acknowledged
4	Duplicated logic for calculating underlying supply	Best Practices	Acknowledged

6 System Overview

The **AlphaHYPE** (αHYPE) token represents the liquid staking token (LST) on the **HyperEVM**. All core functionalities of the protocol are implemented within a single contract — `AlphaHYPEManager03`. This contract is responsible for handling **deposits**, **withdrawals**, and all interactions with the **HyperCore** staking layer.

Deposits

Deposits are handled via the internal logic of the `receive()` and `fallback()` functions. Through these functions, native `HYPE` tokens are sent to the manager contract, where each deposit is added to the **deposit queue** for later processing.

Before being accepted, several internal checks are performed:

- Validation of precision rounding (as HyperCore assets use 10^8 decimals),
- Enforcement of a minimum deposit amount to prevent queue spam,
- Limiting the number of pending deposits to a maximum of 100.

Withdrawals

Once a user holds αHYPE tokens, they can redeem them by initiating a withdrawal request using:

```
function withdraw(uint256 _amount) external nonReentrant;
```

During this call, the system calculates the **current price** based on the total HYPE balance held within the protocol and the total supply of αHYPE tokens. This price is then used to determine the withdrawal amount, which is stored in the **withdrawal queue**.

Queue Processing

All pending requests are processed via the key system function:

```
function processQueues() external nonReentrant;
```

This function first determines the current exchange price for deposits and withdrawals (both calculated the same way, differing only in rounding direction). Using this price:

- The **deposit queue** is processed by minting an appropriate amount of αHYPE tokens to users.
- The **withdrawal queue** is then processed by redeeming HYPE tokens for users based on the stored price.

After processing all deposits, the deposited HYPE remains in the contract and may be used to fulfill pending withdrawals. If the new calculated price is lower (e.g., due to slashing), the final redeemable amount is adjusted accordingly.

Interaction with HyperCore

Once queues are processed, two scenarios can occur:

a. **Pending withdrawals remain:** The system attempts to obtain funds from the HyperCore.
   - If assets are available in the HyperCore spot balance, they are transferred back to the manager contract.
   - If funds are undelegated but not yet available, a withdrawal request from the HyperCore staking balance is initiated.
   - If more assets are required, the manager contract undelegates additional tokens from the validator.

b. **All withdrawals fulfilled:** Any remaining idle HYPE balance in the manager contract is deposited back into HyperCore. Spot assets are staked by first depositing them into the staking balance and then delegating them to a validator.

Rewards and Fees

All staking rewards are **automatically compounded** within the system, as this behavior is inherent to the Hyperliquid staking model utilized by the protocol.

Finally, during queue processing, the system applies:

- A **mint fee** on deposits,
- A **burn fee** on processed withdrawals.

7 Issues

7.1 [Medium] Stale precompile data leads to incorrect pricing and loss for withdrawers

File(s): `AlphaHYPEManager03.sol`

Description: The function `getUnderlyingSupply(...)` returns the total HYPE amount held by the protocol. It first checks the native balance and subtracts pending deposits, fees, and claimable withdrawals, as these should not affect the price. Then, the contract interacts with the `DelegatorSummary` precompile, which returns the current state of delegated, undelegated, and pending withdrawal funds (all still considered part of the system). Finally, the spot balance is fetched via another precompile. The issue arises from the behaviour of precompiles. According to the documentation: *The values are guaranteed to match the latest HyperCore state at the time the EVM block is constructed.* *Reference*

This means the precompile values are only updated at the **beginning** of a new block. Therefore, if a transaction interacts with precompiles within the **same block** after HyperCore-related state changes (e.g., assets transferred from HyperEVM to HyperCore), the returned data will be **stale**.

This becomes problematic when the protocol's `processQueues(...)` function is executed to resolve pending deposits and withdrawals. Consider a scenario where only **pending deposits** exist, the following branch of code is executed:

```
if (evmHype8 > 0) {
    uint256 toSendWei = Math.mulDiv(evmHype8, SCALE_18_TO_8, 1);
    (bool success,) = payable(HYPE_SYSTEM_ADDRESS).call{value: toSendWei}("");
    require(success, "Failed to send HYPE to spot");
    emit EVMSend(evmHype8, HYPE_SYSTEM_ADDRESS);
}
if (sb.total > 0) { // @audit likely related to acknowledged L-04 issue
    L1Write.stakingDeposit(sb.total);
    emit StakingDeposit(sb.total);
}
if (ds.undelegated > 0) {
    L1Write.tokenDelegate(validator, ds.undelegated, false);
    emit TokenDelegate(validator, ds.undelegated, false);
}
```

As shown, assets are transferred to HyperCore, affecting the **Hyperliquid spot balance** and reducing the native contract balance. However, since the precompile data is not refreshed mid-block, this new state is **not yet reflected**.

If a user submits a **withdrawal request** in the **same block** as the deposit queue processing, the withdrawal price calculation will use outdated (stale) data. This typically results in an **inflated price** and the **user receiving fewer funds**. The issue occurs because the spot balance used for pricing does not include the newly transferred assets, even though the corresponding `aHYPE` tokens have already been minted.

Although the correct price will be reflected in the **next block**, the user will have already incurred a loss, since the withdrawal process only considers the **current (lower)** price during execution. As a result, the discrepancy never self-corrects.

Impact: Loss of funds for withdrawers when deposits are transferred to HyperCore within the same block.

Recommendation(s): Keep track of the expected precompile state changes if we exist within the same block.

Status: Fixed

Update from aHYPE team: Resolved in cac8c9e457602a9778506d07c09a45894866c00e.

7.2 [Info] HyperCore enforces 5-request withdrawal cap

File(s): AlphaHYPEManager03.sol

Description: Unstaking on HyperLiquid requires two steps:

a. Undelegation — which is immediate if the validator is not locked;
b. Withdrawal from the staking balance — which is subject to a 7-day unstaking period;

Once this period passes, the assets automatically appear in the spot balance on the HyperCore side. However, there is a limitation on unstaking requests in HyperLiquid: a maximum of 5 active withdrawal requests per account. Any additional requests beyond this limit are rejected. The aHYPE manager initiates withdrawal requests when there are insufficient funds in the manager contract to cover user withdrawals and there are assets available in the undelegation balance. In this case, the contract triggers a staking balance withdrawal through the CoreWriter interface on HyperEVM:

```
if (ds.totalPendingWithdrawal < _virtualWithdrawalAmount) {
    // Pending withdrawal doesn't cover withdrawal amount
    _virtualWithdrawalAmount -= ds.totalPendingWithdrawal;
    uint256 toWithdrawFromStaking = Math.min(ds.undelegated, _virtualWithdrawalAmount);
    if (toWithdrawFromStaking > 0) {
        L1Write.stakingWithdraw(toWithdrawFromStaking.toUint64());
        emit StakingWithdraw(toWithdrawFromStaking);
        _virtualWithdrawalAmount -= toWithdrawFromStaking;
    }
    // Not enough pending withdrawals to cover the withdrawals, we need to undelegate the rest
    uint256 toUndelegate = Math.min(ds.delegated, _virtualWithdrawalAmount);
    if (toUndelegate > 0) {
        L1Write.tokenDelegate(validator, toUndelegate, true);
        emit TokenDelegate(validator, toUndelegate, true);
    }
}
```

The issue arises when there are already 5 pending withdrawal requests. In this situation, the contract still attempts to initiate additional withdrawals, but these calls silently fail, resulting in no assets being withdrawn and no error reported.

Impact: Depending on how the front-end handles the event emission, it might cause display issues showing a larger number of pending withdrawals than the actual state.

Recommendation(s): Implement a check to prevent withdrawal attempts from the staking balance when 5 withdrawal requests are already pending. This can be done by verifying the variable that tracks pending requests within the DelegatorSummary.

Status: Acknowledged

7.3 [Info] Undelegation may silently fail due to validator lock

File(s): AlphaHYPEManager03.sol

Description: When unstaking from HyperCore, a user (in this case, AlphaHYPEManager03) must first undelegate their stake. However, if the manager has delegated to the same validator within the past day, there is a 1-day lockup period for undelegation. During the processQueues(...) call, if there are insufficient assets in the manager contract or pending withdrawals, the manager attempts to undelegate funds from the validator using the following code:

```
// Not enough pending withdrawals to cover the withdrawals, we need to undelegate the rest
uint256 toUndelegate = Math.min(ds.delegated, _virtualWithdrawalAmount);
if (toUndelegate > 0) {
    L1Write.tokenDelegate(validator, toUndelegate, true);
    emit TokenDelegate(validator, toUndelegate, true);
}
```

If the validator is locked due to a recent delegation, the call to CoreWriter will silently fail, and no undelegation will occur.

Impact: Depending on how the front-end handles event emissions, this may lead to display inconsistencies, showing more undelegated funds than are actually available.

Recommendation(s): Consider skipping or disallowing this code path when the validator is currently locked to prevent silent failures.

Status: Acknowledged

Update from aHYPE team: Our keeper bot that processes the queue takes into account the unstaking 1d lock.

7.4 [Best Practice] Duplicated logic for calculating underlying supply

File(s): `AlphaHYPEManager03.sol`

Description: The `AlphaHYPEManager03` contract provides a view function `getUnderlyingSupply(...)` that calculates the total underlying assets managed by the contract by summing balances across the EVM, delegator, and spot accounts. The `processQueues(...)` function, which is responsible for processing deposit and withdrawal requests, requires this same `underlyingSupply` value to determine the current price per share. However, instead of calling the existing `getUnderlyingSupply(...)` function, it duplicates the entire calculation logic inside its own function body.

```
function processQueues() external nonReentrant {
    // ...
    uint256 evmHype8 = address(this).balance / SCALE_18_TO_8;

    // ...
    require(
        evmHype8 >= owedUnderlyingAmount + pendingDepositAmount + feeAmount,
        "AlphaHYPEManager: BANKRUPT"
    );

    // @audit The logic below for calculating underlyingSupply is duplicated
    // from the getUnderlyingSupply() function.
    uint256 underlyingSupply =
        evmHype8 - pendingDepositAmount - owedUnderlyingAmount - feeAmount; // EVM
        // balance in 8 decimals

    L1Read.DelegatorSummary memory ds =
        L1Read.delegatorSummary(address(this));

    underlyingSupply +=
        (ds.delegated + ds.undelegated + ds.totalPendingWithdrawal);

    L1Read.SpotBalance memory sb =
        L1Read.spotBalance(address(this), hypeTokenIndex); // Assuming token
        // index 1 for HYPE

    underlyingSupply += sb.total; // Assuming HYPE has 8 decimals
    // ...
}
```

This code duplication is considered bad practice as it increases the contract's size and, more importantly, makes the code harder to maintain. If the logic for calculating the underlying supply ever needs to be changed, it must be updated in both places. Forgetting to update one of them would lead to inconsistencies and potentially critical bugs.

Recommendation(s): Consider refactoring the code to re-use the existing functionality. This will reduce code duplication, improve readability, and make the contract easier to maintain.

Status: Acknowledged

8 Evaluation of Provided Documentation

The **aHYPE** documentation was provided in the form of a README file and NatSpec comments:

- **NatSpec:** The in-code NatSpec comments were generally sufficient and very helpful in explaining specific flows and code branches. In several cases, they also clarified assumptions underlying the implementation, providing context for why certain approaches were taken and the intended behavior of the system.
- **README:** The provided README offered a solid overview of the protocol's functionality. However, in some instances, there were discrepancies between the README and the actual code. Certain sections appeared to be AI-generated and contained misleading or inaccurate points. For example, some descriptions did not accurately reflect the handling of non-slashing risk withdrawals. We strongly recommend reviewing the README and updating or rewriting any sections that do not align with the implementation.

Overall, the documentation was adequate and sufficient for the scope of this audit. Additionally, the αHYPE team remained consistently available and responsive, promptly addressing all questions and concerns raised by **CODESPECT** throughout the audit process.

9 Test Suite Evaluation

9.1 Compilation Output

```
> forge compile
[] Compiling...
[] Compiling 53 files with Solc 0.8.26
[] Solc 0.8.26 finished in 3.23s
Compiler run successful with warnings:
// Opt out
```

9.2 Tests Output

```
> forge test
Ran 3 tests for test/MockPrecompiles.t.sol:MockPrecompiles
[PASS] test_allPrecompiles() (gas: 112360)
[PASS] test_sharedState() (gas: 181603)
[PASS] test_tokenInfo() (gas: 24498)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 9.34ms (3.94ms CPU time)

Ran 8 tests for test/AlphaHYPEManager.security.t.sol:AlphaHYPEManager03SecurityTest
[PASS] test_DepositRoundingDownAtFractionalPrice_MintsFloor() (gas: 426323)
[PASS] test_NoUnderflow_AfterClaim_ProcessQueuesWorks() (gas: 599519)
[PASS] test_Reentrancy_ClaimCannotDoubleClaim() (gas: 990715)
[PASS] test_Reentrancy_ClaimReentersProcessQueues_StaysSafe() (gas: 928632)
[PASS] test_WithdrawRoundingDownAtFractionalPrice() (gas: 588156)
[PASS] test_allPrecompiles() (gas: 112360)
[PASS] test_sharedState() (gas: 181559)
[PASS] test_tokenInfo() (gas: 24498)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 10.06s (12.24ms CPU time)

Ran 36 tests for test/AlphaHYPEManager.t.sol:AlphaHYPEManager03Test
[PASS] test_DelegationWhenNoWithdrawals() (gas: 177189)
[PASS] test_DepositRevertsOnInvalidAmount() (gas: 28501)
[PASS] test_DepositViaFallback() (gas: 122979)
[PASS] test_DepositViaReceive() (gas: 122916)
[PASS] test_InitialState() (gas: 48667)
[PASS] test_InitializeCanOnlyBeCalledOnce() (gas: 19165)
[PASS] test_InitializeRevertsOnZeroValidator() (gas: 5822848)
[PASS] test_MultiUserDepositWithdrawCycle() (gas: 639379)
[PASS] test_MultipleDeposits() (gas: 186241)
[PASS] test_MultipleWithdrawals() (gas: 655424)
[PASS] test_NativeTransferToStakingAddress() (gas: 152572)
[PASS] test_OwnershipAccess() (gas: 40289)
[PASS] test_PriceCalculationWithMixedBalances() (gas: 492875)
[PASS] test_ProcessDepositsInitialPrice() (gas: 266900)
[PASS] test_ProcessDepositsWithExistingSupply() (gas: 442860)
[PASS] test_ProcessPartialWithdrawals() (gas: 907679)
[PASS] test_ProcessQueuesRevertsWhenCalledByNonProcessor() (gas: 307416)
[PASS] test_ProcessQueuesWhenProcessorNotSet() (gas: 259532)
[PASS] test_ProcessQueuesWhenProcessorSet() (gas: 280798)
[PASS] test_ProcessWithdrawalsWithSufficientBalance() (gas: 890531)
[PASS] test_ProcessorChangeAllowsNewProcessorToProceed() (gas: 452997)
[PASS] test_ReentrancyProtection() (gas: 982577)
[PASS] test_RoundingPrevention() (gas: 66855)
[PASS] test_SetProcessor() (gas: 44796)
[PASS] test_SetProcessorRevertsForNonOwner() (gas: 21259)
[PASS] test_SpotToEVMBridgeForWithdrawals() (gas: 460735)
[PASS] test_StakingDepositWhenNoWithdrawals() (gas: 160549)
[PASS] test_StakingPriorityOrder() (gas: 861304)
[PASS] test_UndelegationForWithdrawals() (gas: 536092)
[PASS] test_WithdrawRequest() (gas: 403824)
[PASS] test_WithdrawRevertsOnInsufficientBalance() (gas: 264496)
[PASS] test_WithdrawRevertsOnZeroAmount() (gas: 22240)
[PASS] test_WithdrawalAfterSlash() (gas: 525835)
[PASS] test_allPrecompiles() (gas: 112405)
[PASS] test_sharedState() (gas: 181582)
[PASS] test_tokenInfo() (gas: 24520)
Suite result: ok. 36 passed; 0 failed; 0 skipped; finished in 10.06s (57.25ms CPU time)
```

```
Ran 6 tests for test/AlphaHYPEManager.fuzzing.t.sol:AlphaHypeManagerTest
[PASS] invariant_HypeBalance() (runs: 300, calls: 150000, reverts: 126203)

----------+-----------------+-------+---------+----------
| Contract | Selector        | Calls | Reverts | Discards |
+=====================================================+
| Fuzzer   | claimWithdrawal | 37319 | 37319   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | deposit         | 37734 | 16869   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | processQueues   | 37382 | 37082   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | withdraw        | 37565 | 34933   | 0        |
----------+-----------------+-------+---------+----------

[PASS] invariant_Supply() (runs: 300, calls: 150000, reverts: 126223)

----------+-----------------+-------+---------+----------
| Contract | Selector        | Calls | Reverts | Discards |
+=====================================================+
| Fuzzer   | claimWithdrawal | 37319 | 37319   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | deposit         | 37734 | 16858   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | processQueues   | 37382 | 37082   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | withdraw        | 37565 | 34964   | 0        |
----------+-----------------+-------+---------+----------

[PASS] invariant_VirtualWithdrawalAmount() (runs: 300, calls: 150000, reverts: 126211)

----------+-----------------+-------+---------+----------
| Contract | Selector        | Calls | Reverts | Discards |
+=====================================================+
| Fuzzer   | claimWithdrawal | 37319 | 37319   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | deposit         | 37734 | 16883   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | processQueues   | 37382 | 37082   | 0        |
|----------+-----------------+-------+---------+----------|
| Fuzzer   | withdraw        | 37565 | 34927   | 0        |
----------+-----------------+-------+---------+----------

[PASS] test_allPrecompiles() (gas: 112360)
[PASS] test_sharedState() (gas: 181581)
[PASS] test_tokenInfo() (gas: 24520)
Suite result: ok. 6 passed; 0 failed; 0 skipped; finished in 10.06s (25.93s CPU time)

Ran 4 test suites in 10.07s (30.19s CPU time): 53 tests passed, 0 failed, 0 skipped (53 total tests)
```

9.3 Notes on the Test Suite

The Alpha Hype test suite provides comprehensive coverage for the protocol's primary flows and addresses most common usage scenarios. It is worth noting that the repository provided at the initial audit commit required minor modifications by the audit team to resolve incorrect imports and naming issues before the test suite could be compiled and executed.

While the existing tests are valuable, the primary area for improvement is the invariant testing setup. Most functions called in the Foundry invariant tests currently revert because the fuzzed inputs are improperly bounded, causing them to hit the protocol's standard input validation errors rather than testing deeper state logic. Additionally, the effectiveness of these tests is limited by the HyperLiquid mocks, which do not account for staking rewards and withdrawal time delays. This results in an incorrect assumption of a constant share price, a condition that will not be true in the actual deployment. Expanding the mocks to simulate rewards and refining the invariant test boundaries would significantly enhance the robustness of the test suite.