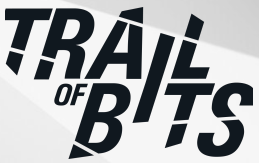

Primitive Hyper

Security Assessment

March 31, 2023

Prepared for:
Alex Angel
Primitive

Prepared by: **Nat Chin, Robert Schneider, Tarun Bansal, and Kurt Willis**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

About Trail of Bits **1**
Notices and Remarks **2**
Table of Contents **3**
Executive Summary **5**
Project Summary **7**
Summary of Recommendations **7**
Project Goals **9**
Project Targets **10**
Project Coverage **11**
Automated Testing **13**
Codebase Maturity Evaluation **24**
Summary of Findings **27**
Detailed Findings **29**
1. Lack of zero-value checks on functions 29
2. Documentation discrepancy in computePriceWithChangeInTau 30
3. Risk of token theft due to possible integer underflow in slt 32
4. Risk of token theft due to unchecked type conversion 34
5. Users can swap without paying any fees 36
6. Swap function returns incorrectly scaled output token amount 38
7. Liquidity providers can withdraw total fees earned by a pool 40
8. Asset token price deviates from the price curve of the pool 42
9. New pair creation can overwrite existing pairs 43
10. Error in Invariant.getX 45
11. Pools with overflowing maturity dates can be created 46
12. Minting funds to the Hyper contract arbitrarily increases the next caller's balance 48
13. Pool strike price could be zero due to lack of lower bound check on maxTick 49
14. Rounding error allows liquidity to be added without depositing tokens 51
15. Attackers can sandwich changeParameters calls to steal funds 53
16. Limited precision in strike prices due to fixed tick spacing 57
17. Functions that round by adding 1 result in unexpected behavior 59
18. Solidity compiler optimizations can be problematic 61
19. getAmountOut returns incorrect value when called by controller 62

20. Mismatched base unit comparison can inflate limit tolerance 64
21. Incorrect implementation of edge cases in getY function 66
22. Lack of proper bound handling for solstat functions 68
23. Attackers can steal funds by swapping in both directions 71
A. Vulnerability Categories 73
B. Code Maturity Categories 75
C. Rounding Recommendations 77
D. Risks with Arbitrary Tokens and Third-Party Controllers 79
E. Recommendations for Overflow and Underflow Analysis in Assembly Blocks 80
F. Staking Issues 82
G. Code Quality Recommendations 84

Executive Summary

Engagement Overview

Primitive engaged Trail of Bits to review the security of its Hyper smart contracts. From January 3 to January 31, 2023, a team of four consultants conducted a security review of the client-provided source code, with eight person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	9
Medium	2
Low	4
Informational	6
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Configuration	1
Data Validation	11
Undefined Behavior	11

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **Missing data validation throughout the codebase**
  Throughout the codebase, many functions fail to check that incoming user arguments are bound between two values, resulting in the ambiguity of expected input parameters. For example, missing checks on type conversions could allow attackers to steal funds (TOB-HYPR-4), and pools could be defined with a zero strike price (TOB-HYPR-13).

- **Incorrect handling of arithmetic**
  The system is also affected by the mishandling of arithmetic operations, which could allow attackers to steal funds. These issues stem from underlying assumptions that cause integer overflow issues (TOB-HYPR-3) and rounding issues (TOB-HYPR-13, TOB-HYPR-14, TOB-HYPR-16, TOB-HYPR-17, TOB-HYPR-20, TOB-HYPR-22, and TOB-HYPR-23). We recommend that the Primitive team continue to extend the existing Echidna suite to test for these corner cases in unexpected behavior.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Nat Chin, Consultant
natalie.chin@trailofbits.com

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
January 3, 2023	Pre-project kickoff call
January 9, 2023	Status update meeting #1
January 17, 2023	Status update meeting #2
January 24, 2023	Status update meeting #3
February 2, 2023	Delivery of report draft
February 2, 2023	Report readout meeting
March 31, 2023	Delivery of final report

Summary of Recommendations

Trail of Bits recommends that Primitive address the findings detailed in this report and take the following additional steps prior to deployment:

- Perform additional economic analysis on the Hyper curve to ensure that input and output bounds are always known and checked. Add these bounds into the Hyper smart contracts to ensure that all functions behave as expected.
- Identify additional global system invariants and continue to extend the Echidna end-to-end test suite to ensure that it captures all functions. Run and maintain this extended fuzzing campaign on a server with a corpus.
- Document all implicit and explicit uses of rounding in the system. Integrate this documentation into a flowchart to create a visual representation outlining all of the formulas' rounding directions to ensure that they always favor pools over users.
- Follow all of the recommendations related to rounding outlined in appendix C.
- Perform another security review on a release candidate prior to production deployment after all of the issues in this report have been addressed and fixed.

Project Goals

The engagement was scoped to provide a security assessment of Primitive's Hyper smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker steal funds?
- Are there appropriate access control measures in place for users and admins?
- Does the system's behavior match the specification?
- Can an attacker freeze funds or deny service to the system?
- Is it possible to perform system operations without paying the required fees?
- Are the arithmetic libraries used correctly and do they correctly apply rounding directions?

Project Targets

The engagement involved a review and testing of the targets listed below.

hyper

Repository	https://github.com/primitivefinance/hyper
Versions	`0bcadb708272276dd77a99f58e57f9f8dfed3c79` (Findings 1–17) `577b5c861a9541a91ec39b3618278f93869d0d38` (Findings 19–22) `5a75fc5aad144a1066d9ed19320869c03b305c26`
Type	Solidity
Platform	Ethereum

solstat

Repository	https://github.com/primitivefinance/solstat
Version	`6e9654163765aac867af1a56fc84462ffdad7d56`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches for covering all of the target components involved a combination of static analysis, manual review, and fuzz testing through Echidna:

The `Hyper` contract: The `Hyper` contract is the main entry point for creating pools and using them to deposit and swap tokens. The contract has many features, including the following:

- **Token and pair creation:** Users can define reusable "token pairs" structs and use them to create pools; the structs store the number of token decimals and the address of both tokens. We checked to ensure that pairs can be created and used when pools are created.
- **Pool creation:** Pools are created with a curve to represent a pool with specific parameters. These pools can be set up as mutable or immutable, which determines whether a specified "pool controller" can later change the pool parameters. We reviewed the pool creation code and implemented a series of global system invariants to check the system's assumptions on created pools.
- **Allocating and unallocating:** This functionality allows users to add and remove funds from a pool. We reviewed the arithmetic and rounding directions used in these operations.
- **Swapping:** This functionality allows users to use pools to swap tokens against their pairs. We reviewed the arithmetic in the contracts to ensure that the calculations round correctly and use the correct scale factors. We reviewed two different versions of the `swap` function; the versions of Hyper that we audited are listed in the "Project Targets" section.

The `HyperLib` contract: This contract manages structs that are used by the `Hyper` contract, including the curve, pair, pools, and orders structs. They provide helper functions to compute amounts, validate parameters, and store state information. We focused on the arithmetic logic in these functions, ensuring that all operations round in the correct direction.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Swapping:** While we spent a significant amount of effort reviewing swaps, this code is incredibly complex. Therefore, we recommend continuing to develop additional fuzz tests against the swapping code to allow fuzzers to explore additional states. Our review of the swapping functionality is inconclusive, and more issues may still be present.

- **Settlement:** The Hyper contracts implement a settlement functionality that iterates over a cached list of tokens to carry out appropriate debits and credits of reserves. This logic is meant to integrate with the jump processing functionality to provide users with a more gas-efficient way to handle tokens. Due to time limitations, we did not review this functionality.

- **The `Enigma` contract and jump processing:** During our audit, we briefly reviewed the functionality of the `Enigma` contract. In the long term, this contract requires more in-depth analysis to ensure that unique identifiers are encoded and decoded correctly, and it requires thorough fuzz testing to catch unexpected behavior.

- **Staking and unstaking:** The Primitive codebase contains functionality to allow users to stake and unstake assets, providing different interest rates and rewards for putting funds into the system. This feature was deprioritized and removed from the scope of this review. The Primitive team later pushed a commit to remove this functionality from the Hyper contracts.

- **Custom function dispatching:** The Hyper contracts implement function custom dispatching, which allows functions to be invoked through the fallback function. Due to time limitations, this functionality was deprioritized and requires additional analysis.

Automated Testing

Trail of Bits has developed unique tools for testing smart contracts. In this assessment, we used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations; for example, Echidna may not randomly generate an edge case that violates a property. We follow a consistent process to maximize the efficacy of testing security properties. When using Echidna, we generate 100,000,000 test cases per property.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation

Test Results

Our automated testing and verification focused on the following system properties:

Global system invariants: Using Echidna, we tested the following expected system properties.

ID	Property	Tool	Result
1	The `locked` variable is always set to 1 outside of execution.	Echidna	**Passed**
2	The Hyper account's `settled` variable is set to `true` immediately after deployment. This variable is saved in this state outside of execution.	Echidna	**Passed**

3	The Hyper account's `prepared` variable is set to `false` outside of execution.	Echidna	**Passed**
4	The number of warm tokens outside of execution is always `0`.	Echidna	**Passed**
5	The priority fee of a controlled pool can never be set to `0`.	Echidna	**Passed**
6	The `Hyper` contract's token balance is always greater than or equal to the reserves of the token saved in the contract.	Echidna	**Passed**
7	A mutable pool can never have a nonzero priority fee.	Echidna	**Passed**
8	A pool's maturity is never less than the last timestamp.	Echidna	**Passed**
9	A pool with a nonzero last price never has zero liquidity.	Echidna	**Passed**
10	When called on a pool with a nonzero `deltaLiquidity` value, the `getLiquidityDeltas` function never returns zero `deltaAsset` or `deltaQuote` amounts.	Echidna	**FAILED (TOB-HYPR-17)**
11	A pool's last price is never greater than the strike price.	Echidna	**FAILED (TOB-HYPR-14)**
12	The strike price of a pool can never be set to `0`.	Echidna	**FAILED (TOB-HYPR-13)**

Pair creation: Using Echidna, we tested the following properties of the `Hyper` contract's state when token pairs are created.

ID	Property	Tool	Result
13	The creation of a token pair with the correct preconditions always succeeds.	Echidna	**Passed**
14	The `pairNonce` variable of the `Hyper` contract increases	Echidna	**Passed**

	when a pair is created.		
15	The same asset and quote token always yield the same pair ID.	Echidna	**Passed**
16	The creation of a token pair with two identical tokens always fails.	Echidna	**Passed**
17	The creation of a token pair with a token that has less than six decimals always fails.	Echidna	**Passed**
18	The creation of a token pair with a token that has more than 18 decimals always fails.	Echidna	**Passed**

Curve creation: Using Echidna, we tested the following properties of the `Hyper` contract's state when curves are created.

ID	Property	Tool	Result
19	A curve's fee can never be set to `0`.	Echidna	**Passed**
20	A curve's priority fee can never exceed the fee.	Echidna	**Passed**
21	A curve's duration can never be set to `0`.	Echidna	**Passed**
22	A curve can never have a volatility greater than the value of `MIN_VOLATILITY`.	Echidna	**Passed**
23	A curve's `createdAt` timestamp can never be set to `0`.	Echidna	**Passed**

Pool creation: Using Echidna, we tested the following properties of the `Hyper` contract's state when pools are created.

ID	Property	Tool	Result
24	The creation of a non-controlled pool with the correct	Echidna	**FAILED**

	preconditions does not revert.		(TOB-HYPR-13)
25	The creation of a non-controlled pool properly sets the pool state to immutable.	Echidna	**Passed**
26	The creation of a non-controlled pool sets the `jit` value to the default value of `JUST_IN_TIME_LIQUIDITY_POLICY`.	Echidna	**Passed**
27	The creation of a controlled pool with the correct preconditions never reverts.	Echidna	**Passed**
28	The creation of a controlled pool sets the pool state to mutable.	Echidna	**Passed**
29	The creation of a pool sets the last timestamp and curve created to the current timestamp.	Echidna	**Passed**
30	The `getVirtualReserves` method always returns values less than the values returned by Hyper's respective `getReserve` function for each token of the pool's pair.	Echidna	**Passed**
31	The `getAmountsWad` method always returns an `amountAssetWad` value that is less than 1e18.	Echidna	**Passed**
32	The `getAmountsWad` method always returns an `amountQuoteWad` value that is less than the return value of `pool.params.strike()`.	Echidna	**Passed**
33	The creation of a controlled pool with a zero-value priority fee always fails.	Echidna	**Passed**

Updating pool parameters: Using Echidna, we tested the following properties of the `Hyper` contract's state when pool parameters are updated.

ID	Property	Tool	Result
34	Updating pool parameters does not update the latest timestamp.	Echidna	**Passed**

35	Updating pool parameters maintains the same controller address.	Echidna	**Passed**
36	Updating pool parameters maintains the same creation time.	Echidna	**Passed**
37	The `priorityFee`, `fee`, `volatility`, `duration`, `jit`, and `maxTick` values are updated as long as they are within the correct bounds.	Echidna	**Passed**

Depositing: Using Echidna, we tested the following properties of the `Hyper` contract's state when users deposit tokens.

ID	Property	Tool	Result
38	Deposits with the correct preconditions always succeed.	Echidna	**Passed**
39	When the `deposit` function is called, the sender's ETH balance decreases by the value of `msg.value`.	Echidna	**Passed**
40	Deposits do not alter the `Hyper` contract's ETH balance.	Echidna	**Passed**
41	Deposits increase the `Hyper` contract's balance of the token being deposited by the amount being deposited.	Echidna	**Passed**
42	Deposits increase the given pool's reserve value for the token being deposited by the amount being deposited.	Echidna	**Passed**

Funding and drawing: Using Echidna, we tested the following properties of the `Hyper` contract's state when users fund and draw tokens from a pool.

ID	Property	Tool	Result
44	Funding operations with the correct preconditions always succeed.	Echidna	**Passed**
45	Calling the `fund` function with insufficient caller funds always fails.	Echidna	**Passed**

46	Calling the `fund` function with insufficient allowance always fails.	Echidna	**Passed**
47	After tokens are funded, the sender's token balance decreases by the funded amount.	Echidna	**Passed**
48	After tokens are funded, the Hyper-registered balance of the user's token increases by the funded amount.	Echidna	**FAILED (TOB-HYPR-12)**
49	After tokens are funded, the reserve balance of the token increases by the funded amount.	Echidna	**FAILED (TOB-HYPR-12)**
50	After tokens are funded, the token balance of the `Hyper` contract increases by the funded amount.	Echidna	**Passed**
51	Calling the `fund` function with zero funds always succeeds.	Echidna	**Passed**
52	Calling the `draw` function with sufficient balance always succeeds.	Echidna	**Passed**
53	Calling the `draw` function with insufficient balance always fails.	Echidna	**Passed**
54	After tokens are drawn, the Hyper-registered balance of the user's token decreases by the drawn amount.	Echidna	**Passed**
55	After tokens are drawn, the saved reserve balance decreases by the drawn amount.	Echidna	**Passed**
56	After tokens are drawn, the recipient's token balance always increases by the drawn amount.	Echidna	**Passed**
56	After tokens are drawn, the `Hyper` contract's token balance decreases by the drawn amount.	Echidna	**Passed**
57	Calling the `draw` function with the zero address as the recipient always fails.	Echidna	**Passed**

58	Funding and drawing operations always result in the same pre- and post-state.	Echidna	**Passed**

Allocating and removing funds: Using Echidna, we tested the following properties of the `Hyper` contract's state when funds are allocated and removed.

ID	Property	Tool	Result
59	Calling the `allocate` function with the correct preconditions always succeeds.	Echidna	**Passed**
60	After funds are allocated, the liquidity for the specified `poolId` increases by the value of `deltaLiquidity`.	Echidna	**Passed**
61	After funds are allocated, the liquidity for the specified `poolId` never decreases.	Echidna	**Passed**
62	After funds are allocated, the `Hyper` contract's reserve token balance for the tokens increases by the values of `deltaAsset` and `deltaQuote` if the caller does not have enough tokens in their balance.	Echidna	**Passed**
63	After funds are allocated, the `Hyper` contract's token balance increases by the values of `deltaAsset` and `deltaQuote`.	Echidna	**Passed**
64	After funds are allocated, the caller's free liquidity position always increases by the value of `deltaLiquidity`.	Echidna	**Passed**
65	After funds are allocated, the physical balance for the asset token increases.	Echidna	**Passed**
66	After funds are allocated, the physical balance for the quote token increases.	Echidna	**Passed**
67	The allocation of funds to a nonexistent pool always reverts.	Echidna	**Passed**
68	The allocation of zero delta liquidity always reverts.	Echidna	**Passed**

69	After funds are allocated, if the fee growth asset pool has changed, the fee growth position is nonzero.	Echidna	**Passed**
70	With the correct preconditions, calls to the `unallocate` function never revert.	Echidna	**Passed**
71	After funds are unallocated, the pool liquidity decreases by the unallocated amount.	Echidna	**Passed**
72	After funds are unallocated, the caller's position liquidity decreases by the unallocated amount.	Echidna	**Passed**
73	The tracked asset reserve balance does not change after funds are unallocated.	Echidna	**Passed**
74	The tracked quote reserve balance does not change after funds are unallocated.	Echidna	**Passed**
75	The physical asset balance of the `Hyper` contract does not change after funds are unallocated.	Echidna	**Passed**
76	When called by a caller without a position, the `unallocate` function reverts.	Echidna	**WIP**
77	The unallocation of funds to a nonexistent pool reverts.	Echidna	**Passed**

Swapping: Using Echidna, we tested the following properties of the `Hyper` contract's state when users swap tokens. Although Echidna passed on the 100,000 fuzzing runs, we recommend continuing to run the test suite with a focus on targeting specific safe input ranges to allow Echidna to easily explore all areas. The properties listed below that require additional adjustments are given the status "WIP".

ID	Property	Tool	Result
78	When called with the correct preconditions, the swap function does not revert.	Echidna	**WIP**
79	Swaps after the given pool reaches maturity always revert.	Echidna	**WIP**
80	Swaps on a nonexistent pool always revert.	Echidna	**WIP**
81	Swaps with a zero amount always revert.	Echidna	**WIP**
82	Swaps with a zero limit amount always revert.	Echidna	**WIP**
83	Swapping an asset token into a pool always decreases the price of the token.*	Echidna	**WIP**
84	Swapping a quote token into a pool always increases the price of the token.*	Echidna	**WIP**
85	Swapping an asset token into a pool increases the token reserves.*	Echidna	**WIP**
86	Prices of the given tokens always change during swap operations.*	Echidna	**WIP**
87	Liquidity does not change during swap operations.*	Echidna	**WIP**
88	Fee growth checkpoints increase during swap operations.*	Echidna	**WIP**
89	A swap in one direction followed by a swap in the opposite direction results in the same state pre- and post-swap.	Echidna	**FAILED (TOB-HYPR-23)**

90	A swap of a quote token with more than 14 decimal places for the asset token reverts.	Echidna	**Passed**

*Invariant tests were written jointly with Primitive throughout the audit.

Math function invariants: Using Echidna, we tested the following properties of the `Hyper` contract's state.

ID	Property	Tool	Result
91	The `expWad` function's output domain is `[0, ∞)`.	Echidna	**Passed**
92	The `expWad` function is strictly monotonically increasing.	Echidna	**Passed**
93	The `expWad` function's maximum error is `0` when `x` is approximately `0`. This is measured as $err := max_{\|x\| < \epsilon}(1 - expWad(x)) * sign(x)$.	Echidna	**Passed**
94	The `sqrt` function is strictly monotonically increasing.	Echidna	**Passed**
95	The `sqrt` function rounds down.	Echidna	**Passed**
96	The `sqrt` function's maximum error is `0`. This is measured as $err := max_{0 < x} \|sqrt(x * x) - x\|$.	Echidna	**Passed**
97	The `pdf` function's output domain is `[0,1/√(2π))`. Note: The output domain is `[0,1/√(2π)]`; however, the upper bound should not be included when rounding correctly. The result is inconsistently rounded up when `x` is `0`.	Echidna	**FAILED** (TOB-HYPR-22)
98	The `pdf` function is monotonically decreasing. Its maximum error on values within its input domain is approximately `0.4e18`. This is measured as $err := max_{x < y} pdf(y) - pdf(x)$.	Echidna	**FAILED** (TOB-HYPR-22)

99	The `pdf` function's maximum error is `0` when `x` is approximately `0`. This is measured as $err := max_{x<\epsilon} pdf(x) - \frac{1}{\sqrt{(2\pi)}}$.	Echidna	**Passed**
100	The `erfc` function's output domain is `[0,2]`.	Echidna	**FAILED** (TOB-HYPR-22)
101	The `erfc` function is monotonically decreasing. Its maximum error on values within its input domain is approximately `1.36e57`. This is measured as $err := max_{x<y} erfc(y) - erfc(x)$.	Echidna	**FAILED** (TOB-HYPR-22)
102	The `erfc` function's maximum error is approximately `3e12` when `x` is approximately `0`. This is measured as $err := max_{x<\epsilon} (erfc(x) - 1) * sign(x)$.	Echidna	**FAILED** (TOB-HYPR-22)
103	The `ierfc` function reverts when given values outside of its input domain. Note: It returns the hard-coded value `+/-1e20` on values outside of its domain, but it should return `+/-` ∞ or, better yet, revert.	Echidna	**FAILED** (TOB-HYPR-22)
104	The `ierfc` function is monotonically decreasing. Its maximum error is approximately `4.9e12` on values within its input domain. This is measured as $err := max_{x<y} ierfc(y) - ierfc(x)$.	Echidna	**FAILED** (TOB-HYPR-22)
105	The `ierfc` function's maximum error is `0` when `x` is approximately `1`.	Echidna	**Passed**

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The Hyper codebase uses a significant amount of complex arithmetic. We found issues related to precision loss, the risk that funds could be trapped, and incorrect rounding directions, which attackers could exploit for profit (TOB-HYPR-3, TOB-HYPR-4, TOB-HYPR-5, TOB-HYPR-6, TOB-HYPR-7, TOB-HYPR-14, TOB-HYPR-16, TOB-HYPR-19, TOB-HYPR-20, TOB-HYPR-22, and TOB-HYPR-23). These areas of the codebase require additional documentation specifying the arithmetic operations' expected behavior and additional invariant-based testing to ensure that they behave as expected.	**Weak**
Auditing	The functions in the Hyper contracts emit sufficient events to help Primitive detect unexpected behavior. A respective event is properly emitted for each state-changing function. However, there are some cases in which users receive obscure underflow/overflow error messages due to the use of assembly operations to check against valid input (appendix G). We recommend writing each event so that it explains and shows the updated state.	**Moderate**
Authentication / Access Controls	Most of the expected access controls across the system are documented. However, documentation on roles and privileges needs to be added to the codebase. Moreover, the access control tests need to be more conclusive to test both happy and unhappy paths.	**Moderate**

Complexity Management	Some functions in the `Hyper` contract are well isolated and easy to test. However, certain functions, such as `allocate` and `unallocate`, use nested helper functions to update the state of pools, reserves, and positions, which make them slightly harder to track. Certain areas in the codebase use assembly over high-level Solidity code, which would benefit from additional documentation. Moreover, the `Enigma` contract and custom function dispatching feature require further analysis.	**Moderate**
Decentralization	Pool controllers have significant control over the system, as they have the ability to update the parameters of a curve. As a result, users have to check that their pools are set up with the values that they intended; and for mutable pools, users have to check that the updated parameters still reflect the option they want to purchase. The core contracts of the system are not upgradeable.	**Moderate**
Documentation	The Hyper contracts contain a significant amount of documentation, which provide a one-to-one cross-reference between the terms and definitions in the white paper and those in the code. The terms related to the system are explained in the white paper through equation derivations, interactive curves through Desmos, and detailed explanations of system components. In many cases, the documentation provided us with sufficient context to understand the complex behavior of the system. However, the contracts lack per-function documentation for complicated logic such as the `swap` function. Additionally, we found minor issues related to incorrect and/or outdated NatSpec comments (TOB-HYPR-2, TOB-HYPR-10); those areas of the codebase should be closely reviewed to ensure they behave as expected. Primitive developed thorough documentation of the codebase's global system invariants that specify the expected state of pools and pairs and per-function preconditions and postconditions that allow us to easily reason about whether the invariants hold. Although many of these invariants are implemented, we recommend that Primitive continue to develop additional	**Moderate**

	documentation on invariants, specifically those related to the adjustment of pool parameters and the tracking of ticks.	
Transaction Ordering Risks	We found one issue that would allow an attacker to profit immediately after a controller has changed parameters of a mutable pool (TOB-HYPR-15). However, additional investigation is required to determine whether transaction ordering poses another risk to the other sections of the Hyper codebase, particularly to the custom function dispatching feature.	**Further Investigation Required**
Low-Level Manipulation	Assembly versions of functions are used in many areas of the system; these versions do not include checks that high-level Solidity versions would have provided (TOB-HYPR-22). Many of these assembly blocks are missing inline comments describing their operations and are missing high-level reference implementations that could be used in differential fuzzing.	**Weak**
Testing and Verification	Unit tests are used for the majority of the codebase; however, the tests are missing unhappy path testing and corner cases in input bounds. Although basic fuzz tests are also present, we recommend that Primitive continue to extend the current end-to-end fuzz tests to catch undesired behavior that could occur due to pools of multiple token decimals and due to complex operations.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lack of zero-value checks on functions	Data Validation	Informational
2	Documentation discrepancy in computePriceWithChangeInTau	Undefined Behavior	Informational
3	Risk of token theft due to possible integer underflow in slt	Data Validation	High
4	Risk of token theft due to unchecked type conversion	Data Validation	High
5	Users can swap without paying any fees	Data Validation	Medium
6	Swap function returns incorrectly scaled output token amount	Data Validation	High
7	Liquidity providers can withdraw total fees earned by a pool	Undefined Behavior	High
8	Asset token price deviates from the price curve of the pool	Undefined Behavior	Undetermined
9	New pair creation can overwrite existing pairs	Undefined Behavior	High
10	Error in Invariant.getX	Undefined Behavior	Informational
11	Pools with overflowing maturity dates can be created	Data Validation	Low

12	Minting funds to the Hyper contract arbitrarily increases the next caller's balance	Configuration	**Informational**
13	Pool strike price could be zero due to lack of lower bound check on maxTick	Data Validation	**High**
14	Rounding error allows liquidity to be added without depositing tokens	Data Validation	**High**
15	Attackers can sandwich changeParameters calls to steal funds	Undefined Behavior	**High**
16	Limited precision in strike prices due to fixed tick spacing	Data Validation	**Low**
17	Functions that round by adding 1 result in unexpected behavior	Undefined Behavior	**Informational**
18	Solidity compiler optimizations can be problematic	Undefined Behavior	**Informational**
19	getAmountOut returns incorrect value when called by controller	Data Validation	**Low**
20	Mismatched base unit comparison can inflate limit tolerance	Data Validation	**Medium**
21	Incorrect implementation of edge cases in getY function	Undefined Behavior	**Low**
22	Lack of proper bound handling for solstat functions	Undefined Behavior	**Undetermined**
23	Attackers can steal funds by swapping in both directions	Undefined Behavior	**High**

Detailed Findings

1. Lack of zero-value checks on functions	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-HYPR-1
Target: `contracts/Hyper.sol`	

Description
Certain setter functions fail to validate incoming arguments; therefore, callers of these functions can accidentally set important state variables to the zero address.

For example, the `constructor` function in the `Hyper` contract, which sets the WETH contract, lacks zero-value checks.

```
constructor(address weth) {
  WETH = weth;
  __account__.settled = true;
}
```

Figure 1.1: The `constructor` function in `Hyper.sol`

If the WETH address is set to the zero address, the admin must redeploy the Hyper contracts to reset the address's value.

Exploit Scenario
Alice deploys a new version of the `Hyper` contract but mistakenly enters the zero address for the WETH address. She must redeploy the system to reset the value of WETH.

Recommendations
Short term, add zero-value checks to all function arguments to ensure that users cannot accidentally set incorrect values, misconfiguring the system.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's continuous integration pipeline, pre-commit hooks, or build scripts.

2. Documentation discrepancy in computePriceWithChangeInTau	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-HYPR-2
Target: `contracts/libraries/Price.sol`	

Description

The formula provided in the `@custom` field of the NatSpec comment for the `computePriceWithChangeInTau` function does not match the formula implemented in the function body. The discrepancy between the documentation and implementation can cause end users to misunderstand what the function actually does.

```
  /**
   * @dev Computes change in price given a change in time in seconds.
   * @param stk WAD
   * @param vol percentage
   * @param prc WAD
   * @param tau seconds
   * @param epsilon seconds
   * @custom:math P(τ - ε) = ( P(τ)^(√(1 - ε/τ)) / K^2 )e^((1/2)(t^2)(√(τ)√(τ- ε) -
(τ - ε)))
   */
```

Figure 2.1: The NatSpec comment for the `computePriceWithChangeInTau` function in `Price.sol`

The function body implements the following formula:

$$P(\tau - \varepsilon) = ((P(\tau)/K)\text{^}(\sqrt{}(1 - \varepsilon/\tau))) * K * e\text{^}((1/2)(\sigma\text{^}2)(\sqrt{}(\tau)\sqrt{}(\tau - \varepsilon) - (\tau - \varepsilon)))$$

The documented and implemented formulas will result in different values for P(τ - ε).

Exploit Scenario

Alice calculates the result of her investment using formulas from the codebase's NatSpec comments. The result of her calculations convinces her to submit transactions. When Alice completes her series of transactions, the end result differs from her expectations.

Recommendations

Short term, correct the formula error in the NatSpec comment for `computePriceWithChangeInTau` so that it matches the implementation.

Long term, thoroughly proofread the NatSpec comments throughout the codebase, especially where they describe important formulas and operations.

3. Risk of token theft due to possible integer underflow in slt

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-3
Target: `contracts/Assembly.sol`	

Description

An attacker can steal funds from a pool using a function that fails to revert on unexpected input.

The `addSignedDelta` function is used by the `allocate`, `unallocate`, and `unstake` functions to alter the liquidity of a pool. These functions lack checks to ensure that the input values are within permissible limits; instead, they rely on the `addSignedDelta` function to revert on unexpected inputs.

The `addSignedDelta` function checks for the sign of the `delta` value; it subtracts its two's complement from the `input` value if `delta` is a negative integer. After the subtraction operation, the code checks for underflow using the `slt` (signed-less-than) function from the EVM dialect of the YUL language. The `slt` function assumes that both arguments are in a two's complement representation of the integer values and checks their signs using the most significant bit. If an underflow occurs in the previous subtraction operation, `slt`'s `output` value will be interpreted as a negative integer with its most significant bit set to 1. Because the `slt` function will find that the `output` value is less than the `input` value based on their signs, it will return 1. However, the `addSignedDelta` function expects the result of `slt` to be `0` when an underflow occurs; therefore, it fails to capture the underflow condition correctly.

```
  function addSignedDelta(uint128 input, int128 delta) pure returns (uint128 output)
{
    bytes memory revertData = abi.encodeWithSelector(InvalidLiquidity.selector);
    assembly {
      switch slt(delta, 0) // delta < 0 ? 1 : 0
      // negative delta
      case 1 {
        output := sub(input, add(not(delta), 1))
        switch slt(output, input) // output < input ? 1 : 0
        case 0 {
          // not less than
          revert(add(32, revertData), mload(revertData)) // 0x1fff9681
        }
      }
```

```
      // position delta
      case 0 {
        output := add(input, delta)
        switch slt(output, input) // (output < input ? 1 : 0) == 0 ? 1 : 0
        case 1 {
          // less than
          revert(add(32, revertData), mload(revertData)) // 0x1fff9681
        }
      }
    }
  }
```

Figure 3.1: The vulnerable `addSignedDelta` function in `Assembly.sol`

Exploit Scenario

Alice allocates 100 USDC into a pool. Eve, an attacker, calls the `unallocate` function with the 100 USDC as the `deltaLiquidity` argument. The `addSignedDelta` function fails to revert on the integer underflow that results, and Eve is able to withdraw Alice's assets.

Recommendations

Short term, take one of the following actions:

- Correct the implementation of the `addSignedDelta` function to account for underflows.
- Use high-level Solidity code instead of assembly code to avoid further issues; assembly code does not support sub 256-bit types.

Regardless of which action is taken, add test cases to verify the correctness of the new implementation; add both unit test cases and fuzz test cases to capture all of the edge cases.

Long term, carefully review the codebase to find assembly code and verify the correctness of these assembly code blocks by adding test cases. Do not rely on certain behavior of assembly code while working with sub 256-bit types because this behavior is not defined and can change at any time.

Note: Do not consider using the `lt` function in place of the `slt` function because it is not sufficient to capture all of the overflow and underflow conditions.

4. Risk of token theft due to unchecked type conversion

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-4
Target: `contracts/Assembly.sol`	

Description

An attacker can steal funds from a pool using a function that fails to revert on unexpected input.

The `addSignedDelta` function is used by the `allocate`, `unallocate`, and `unstake` functions to alter the liquidity of a pool. These functions lack checks to ensure that the input values are within permissible limits; instead, they rely on the `addSignedDelta` function to revert on unexpected inputs.

When the value of `delta` is a positive integer, the result of the `add` function (from the EVM dialect of the YUL language) will overflow; however, the code cannot capture this overflow. This is because the arguments of the function are 128-bit types, but the assembly code does not have types and operates on 256-bit values. The `add` function returns a 256-bit value as its result. The addition of two 128-bit integers can never overflow a 256-bit integer. For this reason, the result of the `add` function will never wrap around the maximum value of a 256-bit integer, and the `slt` function will never find the `output` value to be less than the `input` value, which means it will never return 1 to indicate that an overflow has occurred. As a result, the code fails to capture the overflow condition correctly.

```
  function addSignedDelta(uint128 input, int128 delta) pure returns (uint128 output)
{
    bytes memory revertData = abi.encodeWithSelector(InvalidLiquidity.selector);
    assembly {
      switch slt(delta, 0) // delta < 0 ? 1 : 0
      // negative delta
      case 1 {
        output := sub(input, add(not(delta), 1))
        switch slt(output, input) // output < input ? 1 : 0
        case 0 {
          // not less than
          revert(add(32, revertData), mload(revertData)) // 0x1fff9681
        }
      }
      // position delta
      case 0 {
        output := add(input, delta)
```

```
        switch slt(output, input) // (output < input ? 1 : 0) == 0 ? 1 : 0
        case 1 {
          // less than
          revert(add(32, revertData), mload(revertData)) // 0x1fff9681
        }
      }
    }
  }
```

Figure 4.1: The vulnerable `addSignedDelta` function in `Assembly.sol`

There are multiple ways in which an attacker could use this issue to withdraw more liquidity than they have deposited in a pool.

Exploit Scenario 1

Alice and Bob allocate 500 USDC each into a USDC-ETH pool. The total allocated liquidity is 1,000 USDC. Eve, an attacker, unallocates 1,000 USDC from the entire pool, withdrawing everyone's assets.

Exploit Scenario 2

Alice allocates 100 USDC into a pool. Eve calls the `unstake` function to increase the value of her own liquidity without depositing any assets and then calls `unallocate` to withdraw the funds from the pool.

Recommendations

Short term, take one of the following actions:

- Correct the implementation of the `addSignedDelta` function to account for overflows.
- Use high-level Solidity code instead of assembly code to avoid further issues; assembly code does not support sub 256-bit types.

Regardless of which action is taken, add test cases to verify the correctness of the new implementation; add both unit test cases and fuzz test cases to capture all of the edge cases.

Long term, carefully review the codebase to find assembly code and verify the correctness of these assembly code blocks by adding test cases. Do not rely on certain behavior of assembly code while working with sub 256-bit types because this behavior is not defined and can change at any time.

5. Users can swap without paying any fees

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-5
Target: `contracts/Hyper.sol`	

Description

While the swap function is being processed, the input and output values of the swap are calculated with and without the fee amount assessed from the total. However, if the user's requested amount exceeds the `maxInput` amount, the re-addition of the swap fee to `deltaInput` is missed, allowing the user to swap without paying the fee.

The swap function calls the `_swapExactIn` function, which contains logic to save intermediate values while processing token swaps. The `deltaInput` variable is initially used to derive the `nextIndependent` value without the fee amount assessed. That fee amount should be added back to `deltaInput` afterward so that it can be added to the total amount withdrawn from the user later in the process. The fee amount is added back to `deltaInput` when `_swap.remainder` is less than or equal to `maxInput`, but not if `_swap.remainder` is greater than `maxInput`. If the fee is not added, then when `deltaInput` is added to `_swap.input`, the fee will not be represented in that amount, which means that it will not be withdrawn from the user.

```
  if (_swap.remainder > maxInput) {
    deltaInput = maxInput - _swap.feeAmount;
    nextIndependent = liveIndependent + deltaInput.divWadDown(_swap.liquidity);
    _swap.remainder -= (deltaInput + _swap.feeAmount);
  } else {
    deltaInput = _swap.remainder - _swap.feeAmount;
    nextIndependent = liveIndependent + deltaInput.divWadDown(_swap.liquidity);
    deltaInput = _swap.remainder; // Swap input amount including the fee payment.
    _swap.remainder = 0; // Clear the remainder to zero, as the order has been
filled.
  }

  // Compute the output of the swap by computing the difference between the
dependent reserves.
  if (_state.sell) nextDependent = rmm.getYWithX(nextIndependent);
  else nextDependent = rmm.getXWithY(nextIndependent);

  _swap.input += deltaInput;
  _swap.output += (liveDependent - nextDependent);
```

Figure 5.1: The `_swapExactIn` function in `Hyper.sol`

Exploit Scenario
Eve executes a swap in which the remainder is greater than the maximum input. Due to the calculations in the `swap` function, Eve can swap without paying the fee amount.

Recommendations
Short term, fix the function to factor in `feeAmounts` when the value of `remainder` is greater than the value of `maxInput`.

Long term, thoroughly analyze the system to identify invariants related to proper fee assessment. Fuzz those invariants using Echidna to ensure that the functions return the expected values and that they are accurate.

6. Swap function returns incorrectly scaled output token amount

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-6
Target: `contracts/Hyper.sol`	

Description

The swap function's `output` value is given per unit of liquidity in the given pool, but it is not scaled by the pool's total liquidity. As a result, users will not receive the number of tokens that they expect on swaps:

```
function swap(
  uint64 poolId,
  bool sellAsset,
  uint amount,
  uint limit
) external lock interactions returns (uint output, uint remainder) {
```

Figure 6.1: The function signature of the swap function in `Hyper.sol`

The `output` token value is calculated using the difference between the `liveDependent` and `nextDependent` variables, both of which are calculated using the reserve amount of the input token. However, the output value is not multiplied by the total liquidity value of the pool, so the output amount is scaled incorrectly:

```
_swap.output += (liveDependent - nextDependent);
```

Figure 6.2: The `output` calculation in the `_swapExactIn` function in `Hyper.sol`

This causes the number of output tokens to be either too few or too many, depending on the current amount of liquidity in the pool.

Primitive also discovered this issue during the code review.

Exploit Scenario

Alice swaps WETH for USDC using Hyper. The pool has less than 1 wad of liquidity. The token output value that is returned to Alice is less than what it should be.

Recommendations

Short term, revise the swap function so that it multiplies the `output` token by the total liquidity present in the pool in which the swap takes place.

Long term, identify additional system invariants and fuzz them using Echidna to ensure that the functions return the expected values and that they are accurate.

7. Liquidity providers can withdraw total fees earned by a pool

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-HYPR-7
Target: `contracts/Hyper.sol`	

Description

The `syncPositionFees` function is implemented incorrectly. As a result, liquidity providers can withdraw the total fees earned by a pool and drain assets from the contract reserves.

Pools earn fees from swaps, which should be distributed among the liquidity providers proportional to the liquidity they provided during the swaps. However, the `Hyper` contract instead distributes the total fees earned by the pool to every liquidity provider, resulting in the distribution of more tokens in fees than earned by the pool.

As shown in figure 7.1, the `syncPositionFees` function is used to compute the fee earned by a liquidity provider. This function multiplies the fee earned per wad of liquidity by the `liquidity` value, provided as an argument to the function.

```
function syncPositionFees(
    HyperPosition storage self,
    uint liquidity,
    uint feeGrowthAsset,
    uint feeGrowthQuote
) returns (uint feeAssetEarned, uint feeQuoteEarned) {
    uint checkpointAsset = Assembly.computeCheckpointDistance(feeGrowthAsset,
self.feeGrowthAssetLast);
    uint checkpointQuote = Assembly.computeCheckpointDistance(feeGrowthQuote,
self.feeGrowthQuoteLast);

    feeAssetEarned = FixedPointMathLib.mulWadDown(checkpointAsset, liquidity);
    feeQuoteEarned = FixedPointMathLib.mulWadDown(checkpointQuote, liquidity);

    self.feeGrowthAssetLast = feeGrowthAsset;
    self.feeGrowthQuoteLast = feeGrowthQuote;

    self.tokensOwedAsset += SafeCastLib.safeCastTo128(feeAssetEarned);
    self.tokensOwedQuote += SafeCastLib.safeCastTo128(feeQuoteEarned);
}
```

Figure 7.1: The `syncPositionFees` function in `HyperLib.sol`

The `syncPositionFees` function is used in the `_changeLiquidity` and `claim` functions defined in the `Hyper` contract. In both locations, when the `syncPositionFees` function is called, the value of the pool's total liquidity is provided as the first argument, which is then multiplied by the fee earned per wad of liquidity. The value resulting from the multiplication is then added to the fee earned by the liquidity provider, which means the total fee earned by the pool is added to the fee earned by the liquidity provider.

There are multiple ways in which an attacker could use this issue to withdraw more than what they have earned in fees.

Exploit Scenario 1
Eve provides minimal liquidity to a pool. Eve waits for some time for some swaps to happen. She calls the `claim` function to withdraw the total fee earned by the pool during the period for which Alice and other users have provided liquidity.

Exploit Scenario 2
Eve provides minimal liquidity to a pool using 10 accounts. Eve makes some large swaps to accrue fees in the pool. She then calls the `claim` function from all 10 accounts to withdraw 10 times the total fee she paid for the swaps. She repeats these steps to drain the contract reserves.

Recommendations
Short term, revise the relevant code so that the value of the sum of a liquidity provider's liquidity (`freeLiquidity` summed with `stakedLiquidity`) is passed as an argument to the `syncPositionFees` function instead of the entire pool's liquidity.

Long term, take the following actions:

- In all functions, document their arguments, their meanings, and their usage.
- Add unit test cases that check all happy and unhappy paths.
- Identify additional system invariants and implement Echidna to capture bugs related to them.

8. Asset token price deviates from the price curve of the pool

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-HYPR-8
Target: `contracts/Hyper.sol`	

Description

The asset token price deviates from the price curve of the pool with each swap operation because of the price adjustment performed in the swap function.

On each swap, the `_swapExactIn` function computes new virtual reserves of a pool based on the user's input and then computes the new price of the asset token based on the pool's new virtual reserves. As shown in figure 8.1, a factor of 10^{11} wei is added to the calculated price.

```
_swap.price = (nextPrice * 10_000_001) / 10_000_000;
```

Figure 8.1: The price adjustment statement in the `_swapExactIn` function in `Hyper.sol`

This adjusted price is then used to calculate the output amount in the next swap operation. The price of the asset token increases by the factor of 10^{11} wei with each swap operation, which causes the price of the asset token to deviate from the price curve of the pool.

This adjusted price is also used to calculate the pool's new virtual reserves on subsequent swaps, which means that the virtual reserves of the pool will be different from those used to calculate this price. This creates a difference between the contract's balance of pool tokens and the pool's virtual reserves; this difference increases with every swap operation, causing assets to become stuck in the contract.

Recommendations

Short term, investigate the maximum amount of deviation that can occur between the asset token price and the price curve of the pool to ensure that the error fits within safe bounds.

Long term, execute thorough economic analysis on the implications of any update to system variables. This should include maximum error calculations for all variables.

9. New pair creation can overwrite existing pairs	
Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-HYPR-9
Target: `contracts/Hyper.sol`	

Description

An overflow of the maximum value of `uint24` could occur in the `_createPair()` function, which can be used to overwrite existing pairs.

As shown in figure 9.1, the `_createPair()` function uses an unchecked block to compute and cast the value of `getPairNonce` to assign the value of `pairId`. This `pairId` is used as a key in the `pairs` mapping to store information related to a specific pair.

```
unchecked {
    pairId = uint24(++getPairNonce);
}
```

Figure 9.1: The `pairId` assignment in the `_createPair()` function in `Hyper.sol`

According to the Solidity documentation, values that undergo explicit type conversion are truncated if the new type cannot hold all of the bits required to represent the new value. Here, the type of `getPairNonce` is converted from `uint256` to `uint24`, so if the new value overflows the maximum `uint24` value, the higher-order bits of `getPairNonce` will be truncated to an unexpected value. This means that existing pools will be overwritten by every new pair creation operation, resulting in an inconsistent contract state with the following issues:

- Two sets of assets will store the same `pairId` in the `getPairId` mapping.
- The value of `HyperPair` will be overwritten in the `pairs` mapping.
- All of the previous pools created for the pair will still hold the previous value of `HyperPair`.

This overflow limit may not seem feasible to reach because the maximum value of the `uint24` type is 16,777,215. It would take a lot of time and cost a lot of gas to create so many pairs. However, because of the low-cost nature and higher transaction throughput of the L2 networks, a malicious user could exploit this issue to conduct a DoS attack on the protocol with insignificant financial loss.

We also found that a pool existence check is implemented in `_createPool` function, but an inline comment indicates Primitive's plans to remove this check. We recommend keeping this check to prevent similar `poolId` overflow and overwriting issues.

Exploit Scenario
Eve deploys numerous fake ERC-20 token contracts. She then uses these ERC-20 tokens to create 16,777,215 bogus pairs in the `Hyper` contract. Now, every new pair overwrites existing pairs, making this instance of the protocol unusable.

Recommendations
Short term, make the following changes:

1. Add a check of the `pairId` in `_createPair()` to ensure that it has not already been used to prevent existing pairs from being overwritten.

2. Increase the upper bound of the value of `pairId` by changing its type.

Long term, carefully review the codebase for explicit type conversions. Document scenarios in which these explicit type conversions can result in an overflow or underflow of the result, and use Echidna to test for these scenarios throughout the codebase.

10. Error in Invariant.getX

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-HYPR-10
Target: `Invariant.sol`	

Description

The `getX` function in the `Invariant` contract implements a formula that does not align with the formula specified in the white paper.

```
1     * @dev Computes `x` in `x = 1 - Φ(Φ⁻¹( (y + k) / K ) + σ√τ)`.
```

Figure 10.1: The NatSpec comment for the `getX()` function in `Invariant.sol`

The body of the function matches the formula described in the NatSpec comment of the function. However, the formula itself is derived incorrectly. The actual quantity to be used inside the parentheses should be `(y - k)` instead of `(y + k)`.

The formula is derived by rearranging the terms in the following:

$$y = K\Phi(\Phi{-}1\ (1 - x) - \sigma\sqrt{\tau}) + k$$

By subtracting `k` from both sides of the equation, it becomes clear that the formula should read `(y - k)`.

The actual impact of this error in the current implementation is low because the function is only ever called with the invariant `k = 0`.

Exploit Scenario

In a future release of the protocol, Primitive decides to use the function with the parameter `k != 0`. This breaks the desired invariant after swaps occur.

Recommendations

Short term, correct the `getX` function's code and update the formula in the function's NatSpec comment.

Long term, keep track of the derivations of formulas used throughout the codebase, and add fuzz tests that verify the properties of and assumptions about the functions that implement them.

11. Pools with overflowing maturity dates can be created	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-HYPR-11
Target: `Hyper.sol`	

Description

Users could create pools with maturity date timestamps (which are of type `uint32`) that are too close to when `uint32` timestamps overflow; pools with overflowing timestamps would become unusable.

```
 326    function maturity(HyperCurve memory self) view returns (uint32 endTimestamp)
{
 327        return (Assembly.convertDaysToSeconds(self.duration) +
self.createdAt).safeCastTo32();
 328    }
```

Figure 11.1: The `maturity()` function in `HyperLib.sol`

Maturity dates are currently limited to five years in the future, and the date when `uint32` timestamps will overflow is September 25, 2104.

The maturity date is not validated on pool creation, so pools that will be unusable when the year 2104 approaches could be created. The maturity date is also not validated in the `checkParameters()` function of the `HyperLib` contract, which could allow an attacker to set the timestamp parameter of the pool to an overflowing timestamp.

Exploit Scenario

In a future version of the protocol, Primitive removes the five-year limit on pool maturity dates. Alice, the controller of a pool, decides to set the maturity date past the year 2104 and is able to trap all of the funds of the pool's liquidity providers.

Recommendations

Short term, add a check to the `checkParameters()` function in `HyperLib` to ensure that pools' maturity dates will not overflow.

Long term, thoroughly document all of the assumptions that are made on the codebase's variables. Where types are limited in size, use modular arithmetic or a larger data type to handle any issues that could occur.

12. Minting funds to the Hyper contract arbitrarily increases the next caller's balance

Severity: **Informational**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-HYPR-12
Target: `Invariant.sol`	

Description

When a user mints funds to the `Hyper` contract, the contract relies on a calculation of the difference between its physical balance and its virtual balance of the given token. However, this calculation increases the `Hyper` contract's reserves and the next caller's balance.

To add tokens into the system, users call the `fund` function; this function uses the `_settlement()` function, which calls the `settle()` function. This function uses the `getNetBalance()` function, which calculates the difference between the return value of the `token.balanceOf` function and the `Hyper` contract's reserves of the token.

```
function getNetBalance(AccountSystem storage self, address token, address account)
view returns (int256 net) {
    uint256 internalBalance = self.reserves[token];
    uint256 physicalBalance = __balanceOf__(token, account);
    net = int256(physicalBalance) - int256(internalBalance);
}
```

Figure 12.1: The `getNetBalance()` function in `Hyper.sol`

Exploit Scenario

Eve, an attacker, creates a DRP token. With her minting rights, she mints 1 million DRP to the `Hyper` contract. As a result, when Alice funds her account, the `Hyper` contract's reserve balance and Alice's tracked token balance increase by 1 million DRP, along with the tokens she intended to fund.

Recommendations

Short term, document the fact that the Hyper reserves and the respective user's balance for the airdropped token will be attributed to the next user.

Long term, clearly identify the expected and unexpected flows in the contract to ensure that users are aware of expected behavior.

13. Pool strike price could be zero due to lack of lower bound check on maxTick

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-13
Target: `Hyper.sol`	

Description

The `maxTick` variable is used to approximate the strike price of a pool. However, the code does not validate the lower bound of `maxTick`, which means that the strike price of a pool can be `0`, causing the pool's assets to be mispriced.

The `maxTick` variable is provided by pool creators and is validated on pool creation by the `checkParameters` function, which checks that the `volatility`, `maxTick`, `duration`, `jit`, and `priorityFee` values are within safe bounds. However, for the `maxTick` parameter, the validation function checks only its upper bound:

```
/** @dev Invalid parameters should revert. */
function checkParameters(HyperCurve memory self) view returns (bool, bytes memory) {
    if (!Assembly.isBetween(self.volatility, MIN_VOLATILITY, MAX_VOLATILITY))
        return (false, abi.encodeWithSelector(InvalidVolatility.selector,
self.volatility));
    if (!Assembly.isBetween(self.duration, MIN_DURATION, MAX_DURATION))
        return (false, abi.encodeWithSelector(InvalidDuration.selector,
self.duration));
    if (self.maxTick >= MAX_TICK) return (false,
abi.encodeWithSelector(InvalidTick.selector, self.maxTick)); //  todo: fix, min tick
check?
    if (self.jit > JUST_IN_TIME_MAX) return (false,
abi.encodeWithSelector(InvalidJit.selector, self.jit));
    if (!Assembly.isBetween(self.fee, MIN_FEE, MAX_FEE))
        return (false, abi.encodeWithSelector(InvalidFee.selector, self.fee));
    // 0 priority fee == no controller, impossible to set to zero unless default
from non controlled pools.
    if (!Assembly.isBetween(self.priorityFee, 0, self.fee))
        return (false, abi.encodeWithSelector(InvalidFee.selector,
self.priorityFee));

    return (true, "");
}
```

Figure 13.1: The `checkParameters()` function in `HyperLib.sol`

The `maxTick` value is used to calculate prices, including the strike price of an asset:

```
function strike(HyperCurve memory self) view returns (uint) {
    return Price.computePriceWithTick(self.maxTick);
}
```

Figure 13.2: The `strike()` function in `HyperLib.sol`

However, because the lower bound of `maxTick` is not checked, the `computePriceWithTick` function could return a `0` value for the price, which would cause the system to use the incorrect value for strike prices.

```
/**
 * @dev Computes a price value from a tick key.
 *
 * @custom:math price = e^(ln(1.0001) * tick)
 *
 * @param tick Key of a slot in a price/liquidity grid.
 * @return price WAD Value on a key (tick) value pair of a price grid.
 */
function computePriceWithTick(int24 tick) internal pure returns (uint256 price) {
    int256 tickWad = int256(tick) * int256(FixedPointMathLib.WAD);
    price = uint256(FixedPointMathLib.powWad(TICK_BASE, tickWad));
}
```

Figure 13.3: The `computePriceWithTick()` function in `HyperLib.sol`

Exploit Scenario

Alice creates a pool with a `maxTick` value of `-887272`. Upon calculating the strike price at maturity, the `tickWad` and `price` values are calculated as follows:

$$int256\ tickWad\ =\ -\ 887272\ *\ 1e18\ =\ -\ 7.2e19$$

$$price\ =\ 1\_0001e14\ \hat{}\ (-\ 7.2e19)\ =\ 0$$

Recommendations

Short term, bound `maxTick` to a lower bound that will not allow strike prices to converge to `0`, and have strike prices round up to ensure that they can never be `0`.

Long term, clearly document the expected and unexpected flows in the contract to ensure that users are aware of expected behavior.

14. Rounding error allows liquidity to be added without depositing tokens

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-14
Target: `HyperLib.sol`	

Description

In pools that have an asset token of six decimals, small allocations of those tokens will be scaled by the number of decimals and then rounded down. A `deltaAsset` value of `0` can be returned by the `getLiquidityDeltas` function, even if a nonzero `deltaLiquidity` argument is provided, allowing an attacker to add liquidity without transferring any tokens.

```
function getLiquidityDeltas(
    HyperPool memory self,
    int128 deltaLiquidity
) view returns (uint128 deltaAsset, uint128 deltaQuote) {
    if (deltaLiquidity == 0) return (deltaAsset, deltaQuote);
    (uint amountAsset, uint amountQuote) = self.getAmounts();

    uint delta;
    if (deltaLiquidity > 0) {
        delta = uint128(deltaLiquidity);
        deltaAsset = amountAsset.mulWadUp(delta).safeCastTo128();
        deltaQuote = amountQuote.mulWadUp(delta).safeCastTo128();
    } else {
        delta = uint128(-deltaLiquidity);
        deltaAsset = amountAsset.mulWadDown(delta).safeCastTo128();
        deltaQuote = amountQuote.mulWadDown(delta).safeCastTo128();
    }
}

/** @dev Decimal amounts per WAD of liquidity, rounded down... */
function getAmounts(HyperPool memory self) view returns (uint amountAssetDec, uint
amountQuoteDec) {
    (uint amountAssetWad, uint amountQuoteWad) = self.getAmountsWad();
    amountAssetDec = amountAssetWad.scaleFromWadDown(self.pair.decimalsAsset);
    amountQuoteDec = amountQuoteWad.scaleFromWadDown(self.pair.decimalsQuote);
}
```

Figure 14.1: The `getLiquidityDeltas` and `getAmounts` functions in `HyperLib.sol`

In the calculation of `amountAssetDec` in the `getAmounts` function, the amount of the asset token in wad units (`1e18`) is scaled to a value representative of that token's decimals and then rounded down. If `amountAssetWad` is a small value, `amountAssetDec` is

rounded down to `0` and returned, tricking the system into thinking zero asset tokens are required to fulfill liquidity allocation.

Exploit Scenario

Eve, an attacker, finds or creates a pool where the asset token has six decimals. She calls the `allocate` function and adds the smallest possible unit (1) as the amount to that pool. The `getAmounts` function scales the value to six decimals, rounds down, and returns `0` for `amountAssetDec`, which is then multiplied by `deltaLiquidity`; as a result, `0` is returned for the required `deltaAsset`. The parameters of the pool are changed and the `_increaseReserves` function is called with the correct `deltaQuote` value but `0` for the `deltaAsset` value.

Recommendations

Short term, make one of the following changes:

- Have amounts of token allocations rounded up to the nearest decimal unit depending on the token's assigned decimal value (e.g., for a token with six decimals, amounts should round up to the next token decimal point, which would be `1e12 = 1e(18 - 6)`).
- Add a zero-value check on the return values of `getAmounts`.

Be careful to consider the downstream implications of any short-term fixes implemented for this issue, as the `getAmounts` function is used in critical system operations.

Long term, continue to add unit tests that consider the expected outcomes of a wide array of inputs and scenarios. Document all of the assumptions within the system and implement fuzz testing for them with Echidna in order to catch edge cases like this that might break assumptions that are not readily apparent.

15. Attackers can sandwich changeParameters calls to steal funds

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-HYPR-15
Target: `HyperLib.sol`	

Description

The `changeParameters` function does not adjust the reserves according to the new parameters, which results in a discrepancy between the virtual reserves of a pool and the reserves of the tokens in the `Hyper` contract.

An attacker can create a new controlled pool with the token they want to steal and a fake token. They can then use a sequence of operations—`allocate`, `changeParameters`, and `unallocate`—to steal tokens from the shared reserves of the `Hyper` contract. In the worst-case scenario, an attacker can drain all of the funds from the `Hyper` contract.

The `Hyper` contract does not store the reserves of tokens in a pool. The virtual reserves of the pool are computed with the last price of the given asset token and the curve parameters. Below, we discuss the impact that a change in parameters could have on various operations:

Allocating and Unallocating

A user can add liquidity to or remove liquidity from a pool using the `allocate` and `unallocate` functions. Both of these functions use the `getLiquidityDeltas` function to compute the amount of the asset token and the amount of the quote token required to change the liquidity by the desired amount. The `getLiquidityDeltas` function calls the `getAmountsWad` function to compute the amount of reserves required for adding one unit of liquidity to the pool. The `getAmountsWad` function uses the last price of the asset token (`self.lastPrice`), strike price, time to maturity, and implied volatility to compute the amount of reserves per liquidity.

```
function getAmountsWad(HyperPool memory self) view returns (uint amountAssetWad,
uint amountQuoteWad) {
    Price.RMM memory rmm = self.getRMM();
    amountAssetWad = rmm.getXWithPrice(self.lastPrice);
    amountQuoteWad = rmm.getYWithX(amountAssetWad);
}
```

Figure 15.1: The `getAmountsWad` function in `HyperLib.sol`

When a controller calls the `changeParameters` function, the function updates the pool parameter values stored in the `Hyper` contract to the provided values. These new values are then used in the next execution of the `getAmountsWad` function along with the value of `pool.lastPrice`, which was computed with the previous pool parameters. If `getAmountsWad` uses the previous `pool.lastPrice` value with new pool parameters, it will return new values for the reserves of the pool; however, the `Hyper` contract's token balances will match those computed with the previous pool parameters.

When a user calls the `unallocate` function after a pool's parameters have been updated, the new computed reserve amounts are used to transfer tokens to the user. Because the `Hyper` contract uses shared reserves of tokens for all of the pools, a user can still withdraw the tokens, allowing them to steal tokens from other pools.

Swapping

The swap function computes the price of the asset token using the `_computeSyncedPrice` function. This function uses the pool parameters to compute the price of the asset token. It uses `pool.lastPrice` as the current price of the asset token, as shown in figure 15.2. It then calls the `computePriceChangeWithTime` function with the pool parameters and the time elapsed since the last swap operation.

```
function _computeSyncedPrice(uint64 poolId) internal view returns (uint256 price,
int24 tick, uint updatedTau) {
    HyperPool memory pool = pools[poolId];
    if (!pool.exists()) revert NonExistentPool(poolId);

    (price, tick, updatedTau) = (pool.lastPrice, pool.lastTick,
pool.tau(_blockTimestamp()));

    uint passed = getTimePassed(poolId);
    if (passed > 0) {
        uint256 lastTau = pool.lastTau(); // pool.params.maturity() -
pool.lastTimestamp.
        (price, tick) = pool.computePriceChangeWithTime(lastTau, passed);
    }
}
```

Figure 15.2: The `_computeSyncedPrice()` function in `Hyper.sol`

The `computePriceChangeWithTime` function computes the strike price of the pool and then calls the `Price.computeWithChangeInTau` function to get the current price of the asset token.

```
function computePriceChangeWithTime(
    HyperPool memory self,
    uint timeRemaining,
    uint epsilon
) pure returns (uint price, int24 tick) {
```

```
    uint maxPrice = Price.computePriceWithTick(self.params.maxTick);
    price = Price.computePriceWithChangeInTau(maxPrice, self.params.volatility,
self.lastPrice, timeRemaining, epsilon);
    tick = Price.computeTickWithPrice(price);
}
```

Figure 15.3: The `computePriceChangeWithTime()` function in `HyperLib.sol`

The `computePriceWithChangeInTau()` uses a formula that is derived under the assumption that the implied volatility and strike price of the pool remain constant during the epsilon period. This epsilon period is the time elapsed since the last swap operation. The problem arises when the controller of the pool changes the pool's parameters. The formula used in the `computePriceWithChangeInTau` function then becomes invalid if the epsilon period is greater than zero.

If a user swaps tokens after the controller has updated the curve parameters, then the wrong price computed by the swap function will result in unexpected behavior. This issue can be used by the controller of the pool to swap at a discounted rate.

There are multiple ways in which an attacker could use this issue to steal funds from the `Hyper` contract.

Exploit Scenario 1

Eve creates a new controlled pool with WETH as an asset token and a fake token as a quote token. Eve allocates 1e18 wad of liquidity in the new pool by depositing X amount of WETH and Y amount of the fake token. Eve then doubles the strike price of the pool by calling `changeParameters()`. This change in the strike price changes the virtual reserves of the pool; specifically, it increases the number of asset tokens and decreases the number of quote tokens required to allocate 1e18 wad of liquidity. Eve then unallocates 1e18 wad liquidity and withdraws X1 amount of WETH and Y1 amount of the fake token. The value of X1 is higher than X because of the change in the strike price. This allows Eve to withdraw more WETH than she deposited.

Exploit Scenario 2

Eve creates a controlled pool for two popular tokens. Other users add liquidity to the pool. Eve then changes the pool's parameters to change the strike price to a value in her favor and executes a large swap to benefit from the liquidity added to the pool. The other users see Eve's actions as arbitrage and lose the value of their provided liquidity. Eve has effectively swapped tokens at a discounted rate and stolen funds from the pool's liquidity providers.

Recommendations

Short term, modify the `changeParameters` function so that it computes new token reserve amounts for pools based on updated pool parameters and transfers tokens to or from the controller to align the `Hyper` contract reserves with the new pool reserves.

Long term, carefully review the codebase to find instances in which the assumptions used in formulas become invalid because of user actions; resolve issues arising from the use of invalid formulas. Add fuzz tests to capture such instances in the codebase.

16. Limited precision in strike prices due to fixed tick spacing

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-16
Target: `Price.sol`	

Description

The strike price is allowed to take on values only from a predefined set of values. This is a fixed pricing grid with limited precision, which means that the strike price can deviate from a desired price.

The strike price in Hyper is supplied through an `int24` tick value that maps to a price in the pricing grid, which was computed from the tick base value using the exponential function (`price = TICK_BASE ^ tick`). As a result, the price values are spaced out exponentially from each other.

```
/**
 * @dev Computes a price value from a tick key.
 *
 * @custom:math price = e^(ln(1.0001) * tick)
 *
 * @param tick Key of a slot in a price/liquidity grid.
 * @return price WAD Value on a key (tick) value pair of a price grid.
 */
function computePriceWithTick(int24 tick) internal pure returns (uint256 price) {
    int256 tickWad = int256(tick) * int256(FixedPointMathLib.WAD);
    price = uint256(FixedPointMathLib.powWad(TICK_BASE, tickWad));
}
```

Figure 16.1: The `computePriceWithTick()` function in `Price.sol`

The risk of price deviation is evident when looking at the resulting values from one tick to another. Given a price in the range of 30,000 per quote token (e.g., BTC/USD), the price difference from one tick to another is approximately 3 USD. The difference between the ticks grows exponentially by one part per thousand.

Further imprecisions could compound when computing the next tick from the price after a swap. However, this is not an exploitable issue, as the next tick is not actually used to derive the next price in the protocol.

Exploit Scenario
Alice opens a pool with a USD/BTC pair and wants to set a price of 30,003 USD/BTC. Due to the limited precision in the ticks, the strike price ends up being set to 30,000 USD/BTC.

Recommendations
Short term, document whether this is desired behavior and describe the limitations and rounding issues that could result from it.

Long term, consider whether a fixed price grid is necessary; if it is not, consider using the decimals representation for storing prices.

17. Functions that round by adding 1 result in unexpected behavior

Severity: **Informational**	Difficulty: **Undetermined**
Type: Undefined Behavior	Finding ID: TOB-HYPR-17
Target: `Assembly.sol`	

Description

Unit conversion functions in the `Assembly` contract add 1 to the results of their division operations to round them up, which results in unexpected rounding effects.

The `scaleFromWadUp()` function is used to convert the input for swap operations from a wad unit to a token decimal unit. The function always adds 1 to the result of the division operation, even if the input amount would be a whole number in token decimals. As a result, the user will transfer more tokens than expected.

```
function scaleFromWadUp(uint amountWad, uint decimals) pure returns (uint outputDec)
{
    uint factor = computeScalar(decimals);
    assembly {
        outputDec := add(div(amountWad, factor), 1)
    }
}
```

Figure 17.1: The `scaleFromWadUp()` function in `Assembly.sol`

The `scaleFromWadUpSigned()` also adds 1 to the result of the division operation to round up the return value.

```
function scaleFromWadUpSigned(int amountWad, uint decimals) pure returns (int
outputDec) {
    uint factor = computeScalar(decimals);
    assembly {
        outputDec := add(sdiv(amountWad, factor), 1)
    }
}
```

Figure 17.2: The `scaleFromWadUpSigned()` function in `Assembly.sol`

Recommendations

Short term, use the formula `((a - 1) / b) + 1` in the `scaleFromWadUp()` and `scaleFromWadUpSigned()` functions to compute the rounded up result of the division operation `a/b`.

Long term, review the entire codebase for functions that round to ensure that they do not add 1 unconditionally to results. Add fuzzing test cases to find edge cases that could cause unexpected rounding issues.

References

- Number Logic

18. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-HYPR-18
Target: `solstat/foundry.toml`	

Description
The Hyper contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity `CHANGELOG`. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`— opens up a security vulnerability in the Solstat contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

19. getAmountOut returns incorrect value when called by controller

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-19
Target: `HyperLib.sol`	

Description

When a controller contract calls the `getAmountOut` function, the `feeAmount` should be calculated using the `priorityFee` value; however, the function incorrectly uses the `fee` value. This causes the function to return an incorrect `output` value.

```
  data.feeAmount = ((data.remainder > maxInput ? maxInput : data.remainder) *
self.params.fee) / 10_000;
```

Figure 19.1: The `getAmountOut` function in `HyperLib.sol`

When a controller contract swaps tokens, the fee assessed in the transaction is calculated using the `priorityFee` parameter.

```
  _state.fee = msg.sender == pool.controller ? pool.params.priorityFee :
uint(pool.params.fee);
```

Figure 19.2: The `_swap` function in `Hyper.sol`

The purpose of the `getAmountOut` function is to return the expected token amount that the user would receive after executing a swap. Using the wrong `feeAmount` will skew the `output` calculation, causing a discrepancy between what the user expects to receive and what they actually receive after executing the swap.

Exploit Scenario

Alice wants to swap two tokens from a pool that has a controller contract set. She queries the controller with the proposed swap parameters, and the controller contract calls the `getAmountOut` function with those same parameters. The `getAmountOut` function returns an output amount of five tokens. Alice sends a swap transaction to the controller, which calls the `swap` function on the `Hyper` contract. Only four tokens are returned to Alice instead of the expected five.

Recommendations

Short term, refactor the `getAmountOut` function to accurately mirror calculations made in the `swap` function and to use `priorityFee` when a controller calls `getAmountOut`.

Long term, expand the current unit test suite to ensure that data returned by view functions is accurate and up to date.

20. Mismatched base unit comparison can inflate limit tolerance

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-HYPR-20
Target: `Hyper.sol`	

Description

When a user swaps tokens, the code enforces the user's `limitPrice` value, denominated in the token's decimal units, by comparing it to the value of the pool's `lastPrice` value, denominated in wad units. The discrepancy between these units could prevent a user's intended price limit from being enforced, resulting in a swap at a market rate that the user did not intend to swap at.

When a user calls the swap function, the `limit` argument is used in the `_swapExactIn` function as the `limitPrice` value (figure 20.1); the `limitPrice` value determines whether the user's intended limit price has been exceeded. The user's assumption is that this value is denominated in the quote token's decimal units. The `limitPrice` value is compared to the `nextPrice` value, taken from the pool's `lastPrice` value, and if the user's price limit has been met or exceeded, the swap reverts.

```
uint nextPrice = pools[args.poolId].lastPrice;
if (!sellAsset && nextPrice > limitPrice) revert SwapLimitReached();
if (sellAsset && limitPrice > nextPrice) revert SwapLimitReached();
```

Figure 20.1: The `_swapExactIn` function in `Hyper.sol`

The `lastPrice` variable is denominated in wad units, so it has 18 decimals of precision. If the quote token used to denominate `limitPrice` has any fewer than 18 decimals, then it will always be smaller than intended compared to `nextPrice`. As a result, a swap transaction submitted by a user who has met or exceeded their price limit will not revert as expected.

Exploit Scenario

Alice swaps tokens in a pool in which both tokens have six decimals. She sets a price limit of seven A tokens for one B token. The trade causes the pricing formula to swing to nine A tokens for one B token. The `_swapExactIn` function checks whether 7e6 is greater than 9e18, which it is not. No `SwapLimitReached` error is thrown as a result, and the swap is allowed to complete despite the surpassed price limit.

Recommendations

Short term, modify the associated code so that either the `limitPrice` input value is scaled to wad units or the `lastPrice` value is scaled to however many decimals the quote token has. Clearly document for users the denomination they should use for the price limit.

Long term, expand the current unit test suite to consider token pools with all ranges of decimals; for each scenario, ensure that the `swap` function will revert when a price limit is reached.

21. Incorrect implementation of edge cases in getY function

Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-HYPR-21
Target: `solstat/src/Invariant.sol`	

Description

The `getY` function in the `Invariant` contract deviates from the intended behavior when the value of `R_x` is equivalent to `WAD` and `0`.

```
if (R_x == WAD) return uint256(int256(stk) + inv); // For `ppf(0)` case, because 1 - R_x == 0, and `y = K * 1 + k` simplifies to `y = K + k`
if (R_x == 0) return uint256(inv); // For `ppf(1)` case, because 1 - 0 == 1, and `y = K * 0 + k` simplifies to `y = k`.
```

Figure 21.1: The two incorrect `if` statements in the `getY` function in `Invariant.sol`

The first `if` statement seems to be checking for the `ppf(1)` case, evinced by the comparison of `R_x == WAD`, with `WAD` representing one unit. The function should return the invariant in this case, but it returns the `stk` value summed with the invariant.

```
y = KΦ(Φ⁻¹(1-1) - σ√τ) + k
y = KΦ(Φ⁻¹(0) - σ√τ) + k
y = KΦ(negative infinite - σ√τ) + k
y = KΦ(negative infinite) + k
y = K*0 + k
y = k
```

Figure 21.2: The `getY` derivation from the white paper when `ppf` approaches negative infinity

Similarly, when `R_x` is `0`, the return value should be `stk` summed with the invariant, but it is simply the invariant.

```
y = KΦ(Φ⁻¹(1-0) - σ√τ) + k
y = KΦ(Φ⁻¹(1) - σ√τ) + k
y = KΦ(positive infinite - σ√τ) + k
y = KΦ(positive infinite) + k
y = K*1 + k
y = K + k
```

Figure 21.3: The `getY` derivation from the white paper when approaching positive infinity

Therefore, the return values for the two branches are incorrect.

Exploit Scenario
Alice attempts to execute a swap, for which the `R_x` value is equivalent to `0`. The function returns the invariant rather than `stk` summed with the invariant. This results in further miscalculations in the swaps.

Recommendations
Short term, switch the return value statements in the two affected branches of the `getY` function: if `R_x` is `WAD`, the function should return the invariant, and if `R_x` is `0`, the function should return `stk` summed with the invariant.

Long term, thoroughly document all of the expected edge cases of inputs and check that these edge cases are handled.

22. Lack of proper bound handling for solstat functions

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-HYPR-22
Target: `solstat/Gaussian.sol`	

Description

The use of unchecked assembly across the system combined with a lack of data validation means that obscure bugs that are difficult to track down may be prevalent in the system.

One example of unchecked assembly that could result in bugs is the `getY` function. Due to assembly rounding issues in the codebase, the inverse of the function's two variables does not hold. This function uses targeted functions in the `Gaussian` code. We wrote a fuzz test and ran it on the `getY` function to ensure that the return values are monotonically decreasing. However, the fuzzing campaign found cases in which the `getY` function returns a lower value than it should:

```
Logs:
  Bound Result 999999989999999997
  Bound Result 10000000003
  Bound Result 10000000000
  Bound Result 10000000000
  Bound Result 86400
  Error: a <= b not satisfied [uint]
    Value a: 10000000000
    Value b: 99
```

Figure 22.1: Results of fuzz testing the `getY` function

The solstat `Gaussian` contract contains the cumulative distribution function (`cdf`), which relies on the `erfc` function. The `erfc` function, however, has multiple issues, indicated by its many breaking invariants; for example, it is not monotonic, it returns hard-coded values outside of its input domain, it has inconsistent rounding directions, and it is missing overflow protection (further described below). This means that the `cdf` function's assumption that it always returns a maximum error of `1.2e-7` may not hold under all conditions:

```
/**
 * @notice Approximation of the Cumulative Distribution Function.
 *
 * @dev Equal to `D(x) = 0.5[ 1 + erf((x - μ) / σ√2)]`.
 * Only computes cdf of a distribution with μ = 0 and σ = 1.
```

```
 *
 * @custom:error Maximum error of 1.2e-7.
 * @custom:source https://mathworld.wolfram.com/NormalDistribution.html.
 */
function cdf(int256 x) internal pure returns (int256 z) {
    int256 negated;
    assembly {
        let res := sdiv(mul(x, ONE), SQRT2)
        negated := add(not(res), 1)
    }

    int256 _erfc = erfc(negated);
    assembly {
        z := sdiv(mul(ONE, _erfc), TWO)
    }
}
```

Figure 22.2: The `cdf()` function in `Gaussian.sol`

The `erfc` function (and related functions) are used throughout the `Hyper` contract to compute the contract's reserves, prices, and the invariants. This function contains a few issues, described below the figure.

```
function erfc(int256 input) internal pure returns (int256 output) {
    uint256 z = abs(input);
    int256 t;
    int256 step;
    int256 k;
    assembly {
        let quo := sdiv(mul(z, ONE), TWO)
        let den := add(ONE, quo)
        t := sdiv(SCALAR_SQRD, den)

        // [...]
    }
}
```

Figure 22.3: The `erfc()` function in `Gaussian.sol`

Lack of Overflow Checks

The `erfc` function does not contain overflow checks. In the above assembly block, the first multiplication operation does not check for overflow. Operations performed in an assembly block use unchecked arithmetic by default. If `z`, the absolute value of the input, is larger than ⌈`type(int256).max / 1e18`⌉ (rounded up), the multiplication operation will result in an overflow.

Use of sdiv Instead of div

Additionally, the `erfc` function uses the `sdiv` function instead of the `div` function on the result of the multiplication operation. The `div` function should be used instead because `z` and the product are positive. Even if the result of the previous multiplication operation

does not overflow, if the result is larger than `type(int256).max`, then it will be incorrectly interpreted as a negative number due to the use of `sdiv`.

Because of these issues, the output values could lie well beyond the intended output domain of the function, `[0, 2]`. For example, `erfc(x)` ≈ `1e57` is a possible output value.

Other functions—and those that rely on `erfc`, such as `pdf`, `ierfc`, `cdf`, `ppf`, `getX`, and `getY`—are similarly affected and could produce unexpected results. Some of these issues are further outlined in appendix E.

Due to the high complexity and use of the function throughout this codebase, the exact implications of an incorrect bound on the function are unclear. We specify further areas that require investigation in appendix C; however, Primitive should conduct additional analysis on the precision loss and specificity of solstat functions.

Exploit Scenario

An attacker sees that under a certain swap configuration, the output amount in Hyper's swap function will result in a significant advantage for the attacker.

Recommendations

Short term, rewrite all of the affected code in high-level Solidity with native overflow protection enabled.

Long term, set up sufficient invariant tests using Echidna that can detect these issues in the code. For all functions, perform thorough analysis on the valid input range, document all assumptions, and ensure that all functions revert if the assumptions on the inputs do not hold.

23. Attackers can steal funds by swapping in both directions

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-HYPR-23
Target: `contracts/Hyper.sol`	

Description

Unexpected behavior in the swap function allows users to profit when executing a swap in one direction and then executing the same swap in the other direction.

In a fuzz test, we identified a case in which an attacker is able to create a pool with a certain configuration that allows them to swap in 1 wei of the asset token and then to swap back out a high number of asset tokens. This would allow the attacker to drain the pool. This issue was found toward the end of the audit, so we were unable to locate the root cause.

```
This is the path taken: Swapping: 1 asset -> 1 quote -> 10000000001 asset

Creating Pool:
  controller 0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496
  priorityFee 1
  fee 1
  volatility 100
  duration 272
  jit 0
  stk 3
  price 3

  Allocating liquidity: 171859515069719386357

Selling, then buying asset
Swapping: 1 asset -> 1 quote -> 10000000001 asset
  SWAP dir 0 (selling): asset -> quote
  SWAP dir 1 (buying): quote -> asset
  bal asset 10000000000
  bal quote 0
```

Figure 23.1: The swap output depicting unexpected behavior

Recommendations

Short term, analyze the swap function to identify the root cause of the vulnerability. This issue still persists after fixing overflow issues and unit conversions by replacing assembly code with high-level code.

Long term, add fuzz test cases to find edge cases causing issues with unexpected rounding behavior.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Transaction Reordering Risks	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Rounding Recommendations

Primitive uses fixed-point arithmetic. The current strategy has resulted in incorrect rounding, allowing attackers to benefit from dust and misprices and to steal assets (TOB-HYPR-3, TOB-HYPR-4, TOB-HYPR-5, TOB-HYPR-6, TOB-HYPR-7, TOB-HYPR-14, TOB-HYPR-16, TOB-HYPR-19, TOB-HYPR-20, TOB-HYPR-22, and TOB-HYPR-23). These issues point to a greater need to test the system in more depth. We recommend ensuring that rounding directions always benefit the pool.

Determining Rounding Directions

To determine how to apply rounding (whether up or down), consider the result of the expected output.

For example, the formula for a swap of token x for token y calculates how much of token x must be sent to the contract to receive y.

$$y' = K\phi(\phi^{-1}(1 - x - \sigma\sqrt{\tau}) + k$$

In order to benefit the pool, y' must tend toward a lower value (↘) to minimize the amount paid out. As a result, the following should hold:

- $K\phi(\phi^{-1}(1 - x))$ must round ↘
- $\sigma\sqrt{\tau}$ must round ↗
    - $\sigma\nearrow\sqrt{\tau\nearrow}$
- k must round ↘

Therefore, the mathematics in the formula should perform this check:

$$y'\searrow = K\searrow\phi\searrow(\phi^{-1}(1 - x - \sigma\nearrow\sqrt{\tau\nearrow}) + k\searrow$$

Similar rounding techniques can be applied in all of the system's formulas to ensure that rounding always occurs in the direction that benefits Primitive.

Rounding Results

- When funds leave the pool, these values **should round down** to favor the protocol over the user. Rounding these values up allows attackers to profit from the rounding direction by receiving more than intended on pool interactions.
- When funds enter the pool, these values **should always round up** to maximize the number of tokens a pool receives. Rounding these values down can result in

near-zero values, which allow attackers to profit from the rounding direction by receiving heavily discounted funds. Rounding down to zero can allow attackers to steal funds.

- When fees are calculated, the amount attributed to the fee bucket **should always round up** to maximize the amount the protocol receives. Rounding down means residual dust may be sent to users instead of the protocol.

Recommendations for Further Investigation

- Analyze all instances in which invariants on variable approximations are used, because after rounding and scaling, the original state of the unscaled variable may not be correctly bounded.

- Implement thorough happy and unhappy path testing throughout the codebase.

D. Risks with Arbitrary Tokens and Third-Party Controllers

Primitive aims to allow third-party users to create their own token pairs and pools. These user-created pairs and pools could introduce problems that could allow attackers to steal funds. We recommend that users review the tokens and vet third-party controllers to ensure that pools do not behave unexpectedly.

Ensure that users follow these guidelines when creating token pairs and pools:

- **Pools should never be upgradeable.** Upgradeable pools have inherent risks that may not be apparent with different versions.
- **Tokens should not have a self-destruct capability.** Destructible tokens have inherent risks, including malicious upgrades through `create2`.
- **Users should not be able to change token decimals.** Adjusting a token's decimals to either less than six or greater than 18 will break the token's composability with the arithmetic in the pool. An example is shown in figure D.1.

```
pair_decimals_never_exceed_bounds(uint256): failed!💥
  Call sequence:
    create_pair_with_safe_preconditions(1,0)
    setDecimals(0)
    pair_decimals_never_exceed_bounds(0)
```

Figure D.1: An Echidna failure on a pair whose token decimals changed after creation

- **Tokens should not be interest bearing or re-adjusting.** The formulaic derivation for the AMM relies on a risk-free rate of return for the asset token. Any form of a wrapper token that pays fees to liquidity providers poses risks to the codebase.

E. Recommendations for Overflow and Underflow Analysis in Assembly Blocks

In this appendix, we provide recommendations for improving the assembly blocks in the Primitive Hyper codebase.

Operations in assembly blocks can be problematic if the code does not check for overflows or underflows, and if certain assumptions about the operations' inputs are not documented or checked.

```
function pdf(int256 x) internal pure returns (int256 z) {
    int256 e;
    assembly {
        e := sdiv(mul(add(not(x), 1), x), TWO) // (-x * x) / 2.
    }
    e = FixedPointMathLib.expWad(e);

    assembly {
        z := sdiv(mul(e, ONE), SQRT_2PI)
    }
}
```

Figure E.1: The `pdf()` function in `Gaussian.sol`

For example, the negation operation in the code above (using `mul`, `not`, and `add`), does not check whether `x` equals `type(int256).min`, a possible input to the function. Calling the function with `type(int256).min` would cause an overflow in the addition operation, preventing the result from being negated.

Additionally, the multiplication operation (`-x * x`) could underflow if the result is less than `type(int256).min`.

In the next assembly block in figure E.1, `sdiv` is used, where `div` would be appropriate. The solmate function `expWad`'s maximum output is such that it could be multiplied by one wad (`1e18`) without overflowing in `int256`. Using `sdiv` instead of `div` implies that the result is interpreted as a signed integer. However, `expWad` always outputs positive numbers; therefore, `div` should be used instead.

Because of the way solmate restricts `expWad`'s output, overflow is not an issue in this case. Nonetheless, multiplying `e` by anything larger than `1e18` could result in an overflow, causing `sdiv` to misinterpret an unsigned value as a signed value. These assumptions must be carefully checked and documented when using inline assembly.

Overflow and underflow checks can be omitted via unchecked blocks where appropriate analysis is performed and heavy optimization is required. An example of such analysis is

shown in figure E.2. However, we strongly recommend that overflow checks always be included because it can become hard to keep track of the assumptions when the code evolves.

```
function pdf_checks_overflow(int256 x) internal pure returns (int256 z) {
    uint256 absX = abs(x); // Reverts for `x = type(int256).min`.
    uint256 xSquared = absX * absX; // Overflow check in uint256 is required.
    unchecked {
        // We can safely cast the result of the division to int256, since
        // dividing `xSquared` by `2e18` ensures that the result is less than
`type(int256).max`.
        // The result is positive, which means that a check for `type(int256).min`
        // can be omitted when negating the result.
        x = -int256(xSquared / 2e18);
    }
    int256 e = FixedPointMathLib.expWad(k);

    unchecked {
        // The output of `expWad` is such that it can be safely
        // multiplied by `1e18` without causing an overflow in int256.
        z = e * ONE / SQRT_2PI;
    }
}
```

Figure E.2: An example `pdf_checks_overflow` function with unchecked blocks that contains enough analysis to justify omitting overflow checks

Although overflow checks and appropriate analysis in assembly blocks can improve otherwise unchecked code, a better practice is to use high-level Solidity versions instead, as exemplified in figure E.3. Using high-level Solidity would improve the given function's protection against overflow and underflow (when native overflow and underflow protection is enabled under `pragma ^0.8.0`) and improve the auditability of the code. We recommend that Primitive consider using the higher-level implementations of functions to allow for the use of native in-built protection.

```
function erfc_checks_overflow(int256 input) internal pure returns (int256 output) {
    uint256 z = abs(input); // Reverts for `x = type(int256).min`.
    // We can safely cast the result of the division to int256,
    // because it is positive and less than `type(int256).max`.
    int256 t = int256(1e36 / (1e18 + z / 2));

    int256 step = ERFC_J;

    step = ERFC_I + (t * step / 1e18);
    step = ERFC_H + (t * step / 1e18);
    step = ERFC_G + (t * step / 1e18);
    step = ERFC_F + (t * step / 1e18);
    step = ERFC_E + (t * step / 1e18);
    step = ERFC_D + (t * step / 1e18);
    step = ERFC_C + (t * step / 1e18);
```

```
    step = ERFC_B + (t * step / 1e18);
    step = -ERFC_A + (t * step / 1e18);

    // We can safely cast the result of the division to int256,
    // because it is positive and less than `type(int256).max`,
    // if the multiplication does not revert due to overflow.
    int256 k = step - int256(z * z / 1e18);
    int256 expWad = FixedPointMathLib.expWad(k);
    int256 r = t * expWad / 1e18;

    output = input < 0 ? TWO - r : r;
}
```

Figure E.3: An example `erfc_checks_overflow` function in high-level Solidity

F. Staking Issues

Although the staking-related code was considered out of scope for this audit, we gave these contracts a best-effort review and identified the following issues. Because they were not thoroughly investigated, we recommend that Primitive check the following areas:

- When a swap occurs, the priority fee amount is computed for the total pool liquidity and transferred from the controller. This means that an attacker may be able to profit from executing a swap against the total pool liquidity.
- The `unstakeTimestamp` value is not updated after pool parameters are updated. This may result in undesired behavior.
- In the `claim` and `_changeStake` functions, every liquidity provider gets a fee payment for the total staked liquidity. It is unclear whether this behavior is intended; if it is, it should be thoroughly and clearly documented.
- Users cannot withdraw liquidity after unstaking due to the JIT restriction.

G. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Use consistent naming conventions throughout the codebase.** Choose conventions for naming variables and use those conventions consistently throughout the codebase. Figure G.1 shows an example of variables that use leading underscores and one that does not.

```
uint256 private locked = 1;
Payment[] private _payments;
SwapState private _state;
```

Figure G.1: `private` variable declarations in `Hyper.sol`

- **Beware of potential overflows in assembly blocks.** Information contained in an arbitrarily large bytes array could overflow when loaded into memory.

```
function toBytes32(bytes memory raw) pure returns (bytes32 data) {
     assembly {
         data := mload(add(raw, 32))
         let shift := mul(sub(32, mload(raw)), 8)
         data := shr(shift, data)
     }
 }
```

Figure G.2: The `toBytes32` function in `Assembly.sol`

- **Ensure that variables' higher-order bits are cleared before they are accessed in assembly blocks if their types are less than 256 bits.** Accessing variables of types that are less than 256 bits in assembly blocks does not guarantee that higher-order bits will be zeroed out. See the Solidity documentation for more details on this issue.

```
function addSignedDelta(uint128 input, int128 delta) pure returns (uint128 output) {
    bytes memory revertData = abi.encodeWithSelector(InvalidLiquidity.selector);
    assembly {
        switch slt(delta, 0) // delta < 0 ? 1 : 0
        // negative delta
        case 1 {
            output := sub(input, add(not(delta), 1))
            switch slt(output, input) // output < input ? 1 : 0
            case 0 {
                // not less than
                revert(add(32, revertData), mload(revertData)) // 0x1fff9681
            }
```

```
        }
        // position delta
        case 0 {
            output := add(input, delta)
            switch slt(output, input) // (output < input ? 1 : 0) == 0 ? 1 : 0
            case 1 {
                // less than
                revert(add(32, revertData), mload(revertData)) // 0x1fff9681
            }
        }
    }
}
```

Figure G.3: The `addSignedDelta` function in `Assembly.sol`

- **Avoid casting down inputs.** Accepting inputs of one type only to cast them to a different type in the code can be confusing to end users and can make it difficult to reason about how the system will behave.

```
function allocate(
    uint64 poolId,
    uint amount
) external lock interactions returns (uint deltaAsset, uint deltaQuote) {
    bool useMax = amount == type(uint).max;
    (deltaAsset, deltaQuote) = _allocate(useMax, poolId, (useMax ? 1 :
amount).safeCastTo128());
}
```

Figure G.4: The `allocate` function in `Hyper.sol`

- **Standardize the use of `uint` and `uint256` throughout the code.** `uint` is an alias of `uint256`, and the two can be used interchangeably without altering the underlying type. However, it is best practice to commit to using one or the other throughout a codebase.

- **Follow consistent conventions for outputs returned in tuples.** Doing so can improve the codebase's readability. For example, the code in figure G.5 returns the quote amount first followed by the asset amount; however, the code in figure G.6 outputs them in reverse.

```
function computeReserves(RMM memory args, uint prc) internal pure returns (uint R_y,
uint R_x) {
```

Figure G.5: The `computeReserves` function from `Price.sol` returns the quote amount and then the asset amount.

```
function getLiquidityDeltas(
    HyperPool memory self,
    int128 deltaLiquidity
```

```
) view returns (uint128 deltaAsset, uint128 deltaQuote) {
```

Figure G.6: The `getLiquidityDeltas` function from `HyperLib.sol` returns the asset amount and then the quote amount.

- **Revise certain error messages that lack detail.** In some parts of the codebase, calculations and rounding could trigger underflow and divide-by-zero reverts that do not return helpful error messages. End users could be confused about why exactly their apparently valid transactions have not succeeded. An example of this issue can be found in calculations made by the `computePriceWithChangeInTau` function of the `Price` library.