Uniswap v4 Core

Security Assessment

September 5, 2024

Prepared for:
Alice Henshaw
Uniswap

Prepared by: **Alexander Remie, Priyanka Bose, Benjamin Samuels, Tarun Bansal, Guillermo Larregay, Josselin Feist, and Xiangan He**

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.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
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

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Alexander Remie, Consultant
alexander.remie@trailofbits.com

Benjamin Samuels, Consultant
benjamin.samuels@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Priyanka Bose, Consultant
priyanka.bose@trailofbits.com

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Xiangan He, Consultant
xiangan.he@trailofbits.com

Project Timeline

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

Date	Event
July 15, 2024	Pre-project kickoff call
July 24, 2024	Status update meeting #1
August 5, 2024	Delivery of report draft and readout meeting
September 5, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Uniswap engaged Trail of Bits to review the security of the Uniswap v4 Core smart contracts. Uniswap v4 is the next iteration of the popular decentralized exchange. This fourth iteration builds on the core of Uniswap v3 by adding the following features: a singleton `PoolManager` contract; an optional, arbitrary `Hooks` contract per pool that is called during various stages in the execution; flash accounting; dynamic LP fees; native ETH support; and using ERC6909 for optionally storing user balances internally instead of performing token transfers.

A team of six consultants conducted the review from July 15, 2024 to August 2, 2024, for a total of six engineer-weeks of effort. Our testing efforts focused on the newly introduced features in v4 of the protocol and how those changes affect the existing v3 protocol. Additionally, we focused on the correctness and safety of the numerous usages of inline assembly throughout the codebase. With full access to source code and documentation, we performed static and dynamic testing of the Uniswap v4 core codebase, using automated and manual processes.

Observations and Impact

The Uniswap v4 project is a mature project that builds on top of the Uniswap v3 implementation—in other words, it builds on top of a battle-tested foundation. Every aspect of this project shows that the Uniswap v4 team has put a lot of effort into its security. The code quality—of the high-level Solidity as well as the low-level inline assembly—is high; there is sufficient source-level documentation; there is proper high-level, user-facing documentation with diagrams and use cases; and the testing suite is extensive and uses fuzzing.

The extensive testing suite heavily uses foundry fuzzing to extend the tested values, leading to higher overall testing coverage. Forge test coverage is integrated in the CI and displayed in each PR. Echidna tests are present to test out the various math libraries.

Since the implementation is significantly geared toward gas optimization, many parts have been implemented in inline assembly instead of Solidity. This has made the implementation significantly more difficult to comprehend and presents many more possible footguns for (future) Uniswap developers. However, while the inline comments of the assembly code help to lower the complexity, the lack of a Solidity counterpart and the level of assembly usage increase the risks.

The newly added Hooks feature enables extensive customization by external projects that previously would have had to fork Uniswap v3 to create their customized version. In v4, these projects can use the Hooks feature to create a customized pool to their liking.

However, the flipside of this flexibility is the added complexity of the overall Uniswap v4 project compared to v3. This flexibility also makes it more difficult for users to choose a non-malicious pool, since they must confront challenges like deciding which pool to use and determining that it is not malicious.

During this review, we found six issues, of which one is low severity and the remaining five are informational. In addition, we created 100 invariants, of which eight have been formally verified by Halmos, 88 checked by Medusa and Echidna, and four statically checked by Slither. See appendix D for an in-depth overview of how we used fuzzing during this engagement.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Uniswap take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Add more documentation throughout the source code.** This will help prevent (new) developers from introducing mistakes because they forget undocumented assumptions or misunderstand the complex and assembly-heavy implementation.
- **Reuse and potentially expand the provided stateful invariants and fuzzing harnesses that we developed during this engagement.** This allows the Uniswap team to easily perform more advanced stateful fuzzing that supersedes what can be done using Foundry. The provided stateful invariants can also easily be extended with new invariants, for which we provide some recommendations in the Future Work section.
- **Write extensive guidelines for users that help them avoid being tricked into using a malicious pool.** As highlighted in TOB-UNI4-6, it is much easier in Uniswap v4 than in Uniswap v3 for malicious actors to trick users into using malicious pools. Additionally, it is generally easier to create malicious pools in Uniswap v4 due to the arbitrary nature of hooks contracts. Guidelines can significantly lower these risks.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	1
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Auditing and Logging	1
Data Validation	3
Undefined Behavior	2

Project Goals

The engagement was scoped to provide a security assessment of the Uniswap v4 Core contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is there a way to alter the internal accounting of balances or fees without providing tokens or Ether? Can a malicious user generate losses or otherwise gain unauthorized access to funds in pools by providing malicious tokens?
- Can external users or contracts lose funds, receive fewer tokens than expected from a swap, or have their funds stuck in the pool during normal operation?
- Can malicious hooks get access to funds or block operations of a different pool?
- Are access controls correctly implemented? Can malicious users execute privileged functions or change system parameters?
- Are arithmetic operations implemented correctly? For low-level implementations, is it possible to trigger an overflow or unexpected results that can affect pools? Are the low-level functions equivalent to their high-level counterparts? Are rounding directions considered and verified for all critical calculations?
- Do all transient storage manipulation functions clean up the state correctly when the calls return, and if so, can an attacker exploit this? Can a call set a state that is not cleared before returning? Are locks correctly implemented?
- Are storage slots, structures, and data correctly manipulated? Can storage be overwritten by malicious users from a different pool or external calls?
- Are fees and deltas correctly calculated and validated in all usages? Are data structures correctly packed and unpacked, and type casts performed without losing data or precision? Are custom types and user-defined operators correctly implemented?
- Are prices and ticks correctly calculated when liquidity is added and removed or when swaps are performed? Can prices be manipulated?

Project Targets

The engagement involved a review and testing of the following target.

Uniswap v4 core

Repository	https://github.com/Uniswap/v4-core
Version	`7a72031574fc4548ca8fce197114cf87d5a2c037`
Type	Solidity
Platform	EVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **Hooks.** Hooks are a new feature introduced by Uniswap V4 that enable a high level of customization by allowing developers to implement specific functionalities at various stages of a liquidity pool's lifecycle inside an arbitrary `Hooks` contract. During pool creation, an optional `Hooks` contract can be attached to the pool. The lower bits of the `Hooks` contract address determine the enabled/disabled hook calls. There are hooks for pool initialization, adding liquidity, removing liquidity, performing a swap, and donating tokens to LPs.

  We manually reviewed the pool creation process to see if an invalid `Hooks` contract could be configured and if the configured `Hooks` contract address could be changed after pool creation. We used unit tests and manual review of the various lifecycle hooks (swaps, donations, and adding or removing liquidity) to determine if the hook calls are made at the right place in the lifecycle and if, through reentrancy, tokens can be stolen from the protocol (including other pools) or the internal accounting can be incorrectly updated in any way. We reviewed the use of hooks-returning-deltas throughout the protocol to see if the deltas were applied correctly. We checked if an upgradeable `Hooks` contract can in any way lead to unforeseen problems. We verified the implementation of all hook calls within the `Hooks` library and the bitwise operations to enable/disable specific hook calls. We reviewed the `callHook` function to assess its correct functioning, the possibility and effect of a return-bomb attack from the hook, and the use of inline assembly to call and parse the hook call's return value. We used static invariants to assess the use of the `noSelfCall` modifier and whether there are collisions between the functions in the `PoolManager` and the various hooks functions (see Automated Testing: Static Invariants).

- **Flash accounting.** Uniswap v4 introduces flash accounting. This entails keeping an internal delta of tokens owed to a user and tokens owed to the protocol, and only at the very end of all actions is a user required to pay their debt and withdraw their credit. Failing to do both of these things will cause the transaction to revert. For a user to pay their debt, they can call `settle` (or burn `ERC69096` tokens through `burn`), and for a user to withdraw their credit, they can call `take` (or mint `ERC6909` tokens through `mint`).

  We manually reviewed the delta mechanism throughout the various lifecycle functions. We looked for ways to bring the delta to zero without paying the full debt, such as paying the debt in a different token instead of the expected one; flaws due to multiple/reentrant calls to `sync`, `settle`, and `take` in various orders; and ways to

maliciously inflate the credit delta amount or set credit in a different token than the expected one. We manually reviewed the delta libraries and types (`BalanceDelta`, `BalanceSwapDelta`, `CurrencyDelta`, `NonZeroDeltaCount`) for flaws due to the use of inline assembly, specifically the correct use of arithmetic-related inline assembly operations.

We also tested flash accounting using stateful invariant testing (UNI-SETTLE-1 through UNI-SETTLE-13).

- **(Dynamic) LP fees.** Whereas Uniswap v3 had a limited set of possible LP fee tiers, Uniswap v4 allows a pool to set an arbitrary LP fee up to 100%. A pool can choose to set a static LP fee or a dynamic LP fee during pool creation. In the case of a dynamic LP fee, the pool's configured `Hooks` contract can at any point update the LP fee by calling `updateDynamicLPFee`.

  We manually reviewed the `updateDynamicLPFee` function, `LPFeeLibrary`, and the LP fee-related validation inside the `Hooks.initialize` function to look for ways to configure an invalid LP fee. We also reviewed the correct application of the static and dynamic LP fee in the `swap` function.

- **Protocol fees.** An optional protocol fee can be enabled by the Uniswap team through a configured `ProtocolFeeController` contract.

  We manually reviewed the access controls on the functions that update and collect the protocol fee. We looked for ways in which a malicious `ProtocolFeeController` contract could lead to a DoS. We also reviewed the use of inline assembly (TOB-UNI4-4) and the correct handling of the low-level call result. We reviewed the effect of withdrawing the protocol fee while the contract is unlocked (TOB-UNI4-3).

  We also tested protocol fees using stateful invariant testing, focusing on assessing whether the singleton contract always contains enough funds to cover outstanding protocol fee debts (UNI-ACTION-6, UNI-ACTION-3).

- **Native ETH support.** Uniswap v4 added back support for native ETH instead of only WETH.

  We verified that ETH is correctly handled as a particular case in all transfer-related functions. We looked for a way to use ETH to settle an outstanding ERC20 debit and vice versa.

- **Singleton pool contract.** The `PoolManager` in Uniswap v4 is a crucial component and provides an entrypoint to the protocol. It maintains the state of the pools and incorporates lifecycle functions like `initialize`, which configures a new pool;

`swap`, which facilitates the exchange of currencies within a pool; and `modifyLiquidity`, which allows modifications in the liquidity provided. The balance functions consist of `mint` and `burn`, exclusively dealing with the creation and destruction of `ERC6909` claims; `take`, for withdrawing a specific amount of currency; and `settle`, for compensating outstanding balances.

We manually reviewed whether the pool initialization is done correctly; whether critical storage variables are updated following an untrusted external call that could be abused using reentrancy; and whether the currency deltas for the various actions (swaps, liquidity modifications, mint, take, etc.) are correctly updated. We also examined the potential for an attacker to abuse the pool ID generation (TOB-UNI4-6). Furthermore, we explored whether a malicious user could create a duplicate pool and overwrite the existing liquidity. We also assessed whether the access control checks for the critical functions in the `PoolManager` contract are properly implemented and if sufficient events are emitted (TOB-UNI4-5). We used a static invariant to ensure that only the `settle` and `settleFor` functions are payable (see Automated Testing: Static Invariants). We additionally used extensive stateful invariants to test the various actions in the `PoolManager` (see Automated Testing: Stateful Invariants).

- **ERC6909.** This is an implementation of the ERC6909 proposal (which itself is based on ERC1155, i.e., a multi-token contract). The ERC6909 contract is used to optionally store user balances internally instead of transferring ERC20 tokens. Users can then, later on, use their ERC6909 balance instead of transferring tokens.

  We used manual review to look for common token-related flaws, compared the implementation against the reference implementation, and reviewed the correct updating of account deltas when using the ERC6909 `mint` and `burn` functions within the `PoolManager`.

- **Arithmetic.** Uniswap v4 reuses the math from Uniswap v3. However, the high-level Solidity implementation is (mostly) rewritten in low-level inline assembly (Yul).

  We manually reviewed the math-related libraries, focusing on the differences between Uniswap v3 and Uniswap v4 (i.e., the rewrite into inline assembly). We additionally used stateless fuzzing on various math functionality (see Automated Testing: Stateless Invariants) and differential fuzz testing to compare the `BitMath`, `FullMath`, and `LiquidityMath` libraries (see Automated Testing: Differential testing between v3 and v4). We also used differential testing between the Uniswap inline assembly implementation and a rewrite in high-level Solidity (see Automated Testing: Differential testing between low-level code and high-level implementations).

- **Transient storage.** Uniswap v4 uses transient storage in multiple contracts/libraries (`Exttload`, `CurrencyDelta`, `CurrencyReserves`, and `NonZeroDeltacount`).

Transient storage is used to store data that should persist between different call frames but not beyond the current transaction. Using transient storage for these lowers the gas cost compared to setting and clearing storage variables.

We manually reviewed the correct usage of transient storage, looking for flaws related to: incorrect validation of set values, not correctly masking set values, incorrectly overwriting existing values, not clearing out previously set values, and reusing set transient storage values across multiple call frames that are individual actions.

- **Donation attacks.** We manually reviewed the code to determine whether any donation attacks are present that may cause the fees earned computation to overflow, or alternatively cause the `feeGrowthGlobal` to underflow. We looked at possibilities of manipulating fee growth or currency delta to steal tokens using the `donation` function. We also considered the use of fake tokens to create a malicious pool to manipulate reserves or fees of other legitimate pools using donations. In addition to manual review, we wrote several invariants that would detect potential issues with `feeGrowthGlobal` underflows (UNI-DONATE-1 through UNI-DONATE-9).

Coverage Limitations

The following items were considered out of scope for this engagement:

- Issues previously found in other audits that were not fixed in the target commit of this engagement.
- Issues already known to the Uniswap team and present in a GitHub issue/PR in the Uniswap v4 repo.

Automated Testing

Trail of Bits uses automated techniques to extensively test software's security properties. We use open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

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

Tool	Description	Policy
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	400,000,000 runs, split across four machines with a maximum sequence length of 150. (Approximately 24 hours)
Medusa	A cross-platform go-ethereum-based fuzzer providing parallelized fuzz testing of smart contracts, heavily inspired by Echidna	800,000,000 runs, split across four machines with a maximum sequence length of 150. (Approximately 24 hours)
Slither	A static analyzer platform used to write custom static invariants	No explicitly policy as the rules created run under a few seconds (Appendix E)
Halmos	A symbolic testing tool for EVM smart contracts	Timeout of 2 hours

Summary of Invariants

Component	Invariant Type	Total Number
Pool initialization	Stateful invariants	10
Swap action	Stateful invariants	24
Donate action	Stateful invariants	9
Settlement actions	Stateful invariants	13
Modify liquidity action	Stateful invariants	11
Take action	Stateful invariants	3
End-to-end properties	Stateful invariants	3
Inter-Action properties	Stateful invariants	6
`ProtocolFeeLibrary`	Stateless invariants	2
`LiquidityMath`	Stateless invariants	2
Differential fuzzing between V3 and V4 math libraries	Stateless differential invariants	5
Differential testing between low-level code and high-level implementations	Stateless differential invariants	8
`PoolManager` and `Hooks`	Static invariants	4

Stateful Invariants

Pool Initialization

ID	Property	Result
UNI-INIT-1	`initialize()` should not revert when it is passed a valid set of parameters (tick spacing, price, fee, pool key, non-existing pool).	**Passed**
UNI-INIT-2	`initialize()` must not throw `PoolAlreadyInitialized()` when there is no pre-existing pool initialized with the same `PoolKey`.	**Passed**
UNI-INIT-3	`initialize()` must not throw `InvalidSqrtPrice()` when provided a price within the valid range.	**Passed**
UNI-INIT-4	`initialize()` must not throw `LPFeeTooLarge()` when the fee is in	**Passed**

	the valid range.	
UNI-INIT-5	`initialize()` must not throw `TickSpacingTooLarge()` when the tick spacing is in the valid range.	**Passed**
UNI-INIT-6	`initialize()` must not throw `TickSpacingTooSmall()` when the tick spacing is in the valid range.	**Passed**
UNI-INIT-7	`initialize()` must revert if the provided pool key is already initialized.	**Passed**
UNI-INIT-8	`initialize()` must construct a pool whose tick is greater than or equal to `MIN_TICK`.	**Passed**
UNI-INIT-9	`initialize()` must construct a pool whose tick is less than or equal to `MAX_TICK`.	**Passed**
UNI-INIT-10	`initialize()` must never create a pool with an initial `sqrtPrice` of zero.	**Passed**

Swap Action

ID	Property	Result
UNI-SWAP-1	After a swap, if the pool's active tick did not change, its liquidity must be the same as it was before the swap.	**Passed**
UNI-SWAP-2	The pool's `sqrtPriceX96` should decrease or stay the same after making a `zeroForOne` swap.	**Passed**
UNI-SWAP-3	The pool's `sqrtPriceX96` should increase or stay the same after making a `oneForZero` swap.	**Passed**
UNI-SWAP-4	The pool's new `sqrtPriceX96` must not be lower than the transaction's price limit after making a `zeroForOne` swap.	**Passed**
UNI-SWAP-5	The pool's new sqrtPriceX96 must not exceed the transaction's price limit after making a `oneForZero` swap.	**Passed**
UNI-SWAP-6	The pool's active tick should decrease or stay the same after making a `zeroForOne` swap.	**Passed**
UNI-SWAP-7	The pool's active tick should increase or stay the same after making a `oneForZero` swap.	**Passed**
UNI-SWAP-8	After a `zeroForOne` swap, the fee growth for `currency1` should not change.	**Passed**

UNI-SWAP-9	After a `oneForZero` swap, the fee growth for `currency0` should not change.	**Passed**
UNI-SWAP-10	The swap action must revert if the swap amount is zero.	**Passed**
UNI-SWAP-11	The pool's new price after a swap must be less than `MAX_SQRT_PRICE`.	**Passed**
UNI-SWAP-12	The pool's new price after a swap must be greater than or equal to `MIN_SQRT_PRICE`.	**Passed**
UNI-SWAP-13	The pool's new tick after a swap must be less than or equal to `MAX_TICK`.	**Passed**
UNI-SWAP-14	The pool's new tick after a swap must be greater than or equal to `MIN_TICK`.	**Passed**
UNI-SWAP-15	Swaps respect the `sqrtPriceLimit` ahead of the need to consume `exactInput` or `exactOutput`.	**Passed**
UNI-SWAP-16	For exact input swaps where the price limit is not reached, the `fromBalanceDelta` must match the exact input amount.	**Passed**
UNI-SWAP-17	For exact output swaps where the price limit is not reached, the `toBalanceDelta` must match the exact output amount.	**Passed**
UNI-SWAP-18	If the `fromBalanceDelta` of a swap is zero, the `toBalanceDelta` must also be zero (rounding).	**Passed**
UNI-SWAP-19	For any swap, the amount credited to the user is greater than or equal to zero.	**Passed**
UNI-SWAP-20	For any swap, the amount debited from the user is greater than or equal to zero.	**Passed**
UNI-SWAP-21	For a `zeroForOne` swap, the amount credited to the user must be less than or equal to the total number of tradeable tokens in the pool.	**Passed**
UNI-SWAP-22	For a `oneForZero` swap, the amount credited to the user must be less than or equal to the total number of tradeable tokens in the pool.	**Passed**
UNI-SWAP-23	After a swap, the user's `currencyDelta` for `amount0` should match the expected delta based on `BalanceDelta`.	**Passed**
UNI-SWAP-24	After a swap, the user's `currencyDelta` for `amount1` should match the expected delta based on `BalanceDelta`.	**Passed**

Donate Action

ID	Property	Result
UNI-DONATE-1	After a donation with a non-zero `amount0`, the pool's `feeGrowthGlobal0X128` should match the expected value based on `donate()`'s inputs.	**Passed**
UNI-DONATE-2	After a donation with a non-zero `amount1`, the pool's `feeGrowthGlobal1X128` should match the expected value based on `donate()`'s inputs.	**Passed**
UNI-DONATE-3	After a donation with a zero amount0, the pool's `feeGrowthGlobal0X128` should not change.	**Passed**
UNI-DONATE-4	After a donation with a zero amount1, the pool's `feeGrowthGlobal1X128` should not change.	**Passed**
UNI-DONATE-5	Donating to a pool with zero liquidity should result in a revert.	**Passed**
UNI-DONATE-6	A `donate()` call must not return a positive `BalanceDelta` for `currency0`.	**Passed**
UNI-DONATE-7	A `donate()` call must not return a positive `BalanceDelta` for `currency1`.	**Passed**
UNI-DONATE-8	The `donate()` call `BalanceDelta` must match the amount donated for `amount0`.	**Passed**
UNI-DONATE-9	The `donate()` call `BalanceDelta` must match the amount donated for `amount1`.	**Passed**

Settlement Actions

ID	Property	Result
UNI-SETTLE-1	The user must not be owed more tokens after a `settle()` or `settleFor()` than they were owed before the settlement.	**Passed**
UNI-SETTLE-2	The amount paid during a `settle()` or `settleFor()` must equal the difference in the user's currency deltas before and after the `settle()` call.	**Passed**
UNI-SETTLE-3	The amount paid during a `settle()` or `settleFor()` must equal the remittances paid to the singleton.	**Passed**
UNI-SETTLE-4	After a burn, the sender's currency delta should increase to reflect the decreased debt. (Weak)	**Passed**

UNI-SETTLE-5	After a burn, the difference between the sender's previous and new currency delta should equal the burn amount. This is a strong version of UNI-SETTLE-4.	**Passed**
UNI-SETTLE-6	After a burn, the `from` actor's ERC6909 balance should decrease to reflect the burned amount. (Weak)	**Passed**
UNI-SETTLE-7	After a burn, the difference between the `from` actor's previous and new ERC6909 balance should equal the burn amount. This is a strong version of UNI-SETTLE-6.	**Passed**
UNI-SETTLE-8	After a mint, the sender's currency delta should decrease to reflect increased debt. (Weak)	**Passed**
UNI-SETTLE-9	After a mint, the difference between the sender's previous and new currency delta should match the mint amount. This is a strong version of UNI-SETTLE-8.	**Passed**
UNI-SETTLE-10	After a mint, the recipient's ERC6909 balance should increase. (Weak)	**Passed**
UNI-SETTLE-11	After a mint, the recipient's ERC6909 balance should increase by the mint amount This is a strong version of UNI-SETTLE-10.	**Passed**
UNI-SETTLE-12	After a clear, the actor's currency delta should go down or be equal to zero. (Weak)	**Passed**
UNI-SETTLE-13	After a clear, the actor's currency delta should equal the amount cleared. This is a strong version of UNI-SETTLE-12.	**Passed**

Modify Liquidity Action

ID	Property	Result
UNI-MODLIQ-1	For a specific position, `getPositionInfo` must return the same liquidity as `getPosition`.	**Passed**
UNI-MODLIQ-2	For a specific position, `getPositionInfo` must return the same `feeGrowthInside0` as `getPosition`.	**Passed**
UNI-MODLIQ-3	For a specific position, `getPositionInfo` must return the same `feeGrowthInside1` as `getPosition`.	**Passed**
UNI-MODLIQ-4	The `amount0` of fees accrued from `modifyPosition()` must be non-negative.	**Passed**
UNI-MODLIQ-5	The `amount1` of fees accrued from `modifyPosition()` must be non-negative.	**Passed**

UNI-MODLIQ-6	The singleton must be able to credit the user for their `feesAccrued.amount0`.	**Passed**
UNI-MODLIQ-7	The singleton must be able to credit the user for their `feesAccrued.amount1`.	**Passed**
UNI-MODLIQ-8	The pool must have enough `currency0` to return the LP's liquidity balance.	**Passed**
UNI-MODLIQ-9	The pool must have enough `currency1` to return the LP's liquidity balance.	**Passed**
UNI-MODLIQ-10	The singleton must have enough `currency0` to return the LP's liquidity balance.	**Passed**
UNI-MODLIQ-11	The singleton must have enough `currency1` to return the LP's liquidity balance.	**Passed**

Take Action

ID	Property	Result
UNI-TAKE-1	After executing `take()`, the user's `currencyDelta` should be the difference between their previous delta and the amount taken.	**Passed**
UNI-TAKE-2	After executing `take()`, the user's balance should increase by the amount taken.	**Passed**
UNI-TAKE-3	After executing `take()`, the singleton's balance should decrease by the amount taken.	**Passed**

End-to-End Properties

ID	Property	Result
UNI-E2E-1	Outstanding deltas must be zero after the singleton is re-locked.	**Passed**
UNI-E2E-2	When swapping through a pair in one direction, then swapping in the opposite direction, the user must not have more `fromTokens` than they started with.	**Passed**
UNI-E2E-3	When swapping through a pair in one direction, then swapping in the opposite direction, the user must not have more `toTokens` than they started with.	**Passed**

Inter-Action Properties

ID	Property	Result
UNI-ACTION-1	The amount owed to an actor in a single-actor system must always be less than or equal to the balance of the singleton. (Weak)	Passed
UNI-ACTION-2	The amount owed to an actor in a single-actor system must always be less than or equal to the balance of the singleton, less protocol fees and LP fees. (Strong)	Passed
UNI-ACTION-3	The amount of protocol fees owed may not exceed the singleton's balance (less its deployed liquidity) while the currency has a positive or zero delta.	Passed
UNI-ACTION-4	An actor's debited delta must not exceed `int256.max` for any single action.	Passed
UNI-ACTION-5	An actor's credited delta must not exceed `int256.max` for any single action.	Passed
UNI-ACTION-6	`collectProtocolFees()` must not revert on valid input.	Passed

Stateless Invariants

ProtocolFeeLibrary

ID	Property	Result
UNI-PROTOFEE-1	The swap fee cannot exceed 100%.	Verified
UNI-PROTOFEE-2	ValidProtocolFee is equivalent to `getZeroForOneFee(self) <= MAX_PROTOCOL_FEE && getOneForZeroFee(self) <= MAX_PROTOCOL_FEE`.	Verified

LiquidityMath

ID	Property	Result
UNI-LIQMATH-1	`addDelta` increases/decreases based on y.	Verified
UNI-LIQMATH-2	`addDelta` reverts if it underflows/overflows.	Verified

Differential Testing between V3 and V4

To check the equivalence of the V3 components (Solidity 0.7) against their V4 counterparts (Solidity 0.8), we implemented a harness using a bytecode-based differential testing approach. We compiled the V3 library using Solidity 0.7, and deployed their raw bytecodes in a harness compiled with Solidity 0.8, allowing us to compare both versions.

ID	Property	Result
UNI-DIFFV3-1	`FullMath.mulDiv()` behaves the same on V3 and V4.	**Verified**
UNI-DIFFV3-2 *	`FullMath.mulDivRoundingUp()` behaves the same on V3 and V4.	**Passed (*)**
UNI-DIFFV3-3	`BitMath.mostSignificantBit()` behaves the same on V3 and V4.	**Verified**
UNI-DIFFV3-4	`BitMath.leastSignificantBit()` behaves the same on V3 and V4.	**Verified**
UNI-DIFFV3-5	`LiquidityMath.addDelta` behaves the same on V3 and V4.	**Verified**

(*) UNI-DIFFV3-2 could not be verified by Halmos (timeout two hours) and was checked with Echidna (500,000 calls).

Differential Testing between Low-Level Code and High-Level Implementations

To verify the equivalence between low-level (assembly) gas-optimized implementations and their high-level counterparts, a fuzzing harness was added. Functions in this contract contain snippets of functions using assembly from Uniswap V4, and implementations of the same functions using pure Solidity.

Where possible, the Solidity equivalents were taken from the comments of the assembly implementations, and in the remaining cases, the code was reconstructed from the intended goal of the function.

ID	Property	Result
UNI-LOWLEVEL-1	Low-level implementation of `Position.get()` should match its high-level implementation.	**Passed**
UNI-LOWLEVEL-2	Low-level implementation of `CurrencyDelta.computeSlot()` should match its high-level implementation.	**Passed**
UNI-LOWLEVEL-3	Low-level implementation of `LiquidityMath.addDelta()` should match its high-level implementation.	**Passed**

UNI-LOWLEVEL-4	Low-level implementation of `Pool.tickSpacingToMaxLiquidityPerTick()` should match its high-level implementation.	**Passed**
UNI-LOWLEVEL-5	Low-level implementation of `ProtocolFeeLibrary.isValidProtocolFee()` should match its high-level implementation.	**Passed**
UNI-LOWLEVEL-6	Low-level implementation of `ProtocolFeeLibrary.calculateSwapFee()` should match its high-level implementation.	**Passed**
UNI-LOWLEVEL-7	Low-level implementation of `SwapMath.getSqrtPriceTarget()` should match its high-level implementation.	**Passed**
UNI-LOWLEVEL-8	Low-level implementation of `TickBitmap.compress()` should match its high-level implementation.	**Passed**

Static Invariants

Through the review, we identified multiple code patterns that require to be enforced through the codebase. To ensure their correctness, we wrote a linter tool based on slither.

The following checks were implemented and were all passing:

ID	Property	Why	Result
`noSelfCall_should_not_return`	Functions that use `noSelfCall` do not have any return variable	The modifier is a no-op; any return variable would always be its default value.	**Passed**
`callHook`	Functions that call `callHook` are protected against self-calls	This ensures that the hook will not re-enter to itself.	**Passed**
`pool_manager_function_ids`	`PoolManager`'s functions do not collide with the `Hooks` function	A function collision could lead to setting a hook to be the pool manager itself and executing unexpected code (e.g., having the manager swap assets).	**Passed**
`pool_manager_payable`	Only `settle`/`settleFor` are payable in the pool manager	Only these two functions should receive funds.	**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	New compiler versions are used for the high-level code; therefore, all high-level operations use checked arithmetic by default. The low-level arithmetic is correct, and the results match their high-level counterparts. Rounding directions are considered and explicitly documented. There is widespread usage of unchecked arithmetic to make the code gas-efficient. However, special consideration is made for operations that could overflow or otherwise revert, and edge cases are documented.	**Strong**
Auditing	In general, events are properly emitted for most of the state changes and other critical functions. However, we found a few instances (TOB-UNI4-5) where the event is missing.	**Satisfactory**
Authentication / Access Controls	The protocol fees contract, used to establish and collect the fees belonging to the protocol, is privileged. Access control is implemented. However, there is no two-step procedure to change ownership. There is no indication of multisignature or hardware key usage for the privileged address.	**Satisfactory**
Complexity Management	In general, functions follow the established good practices: they are short, have little or no redundancy or duplication, and have a clear and limited scope. The naming scheme is also clear. Given the optimizations in the code, assembly code is extensively used, and it can sometimes be quite complex to follow or analyze. Some functions do not validate input parameters to save gas (for example, there are no zero	**Moderate**

	address checks in the code; see Code Quality). There are several usages of custom types and user-defined operators that simplify the understanding of functions and contracts. All these points can impact the onboarding of new developers to the team: the complexity of the system and the low-level optimizations make a steep learning curve, and it is highly likely that an inexperienced developer will introduce bugs inadvertently.	
Decentralization	The protocol is built to be decentralized. The only privileged access functions are the ones related to the protocol fees, which are capped at max 0.1% if they were to be enabled.	**Strong**
Documentation	The documentation clearly shows the differences between V3 and V4 and uses diagrams to summarize information. Reading the documentation can help one gain a high-level understanding of how the different system components in V4 work. Developers integrating with Uniswap V4 have code snippets and examples. Not all contracts are covered in V4 docs. Code documentation is extensive, both in NatSpec and in-line comments. A minor deviation between documentation and code was found in the Hooks library. See the Code Quality appendix.	**Satisfactory**
Low-Level Manipulation	Assembly and low-level structures are used extensively to save gas. Low-level blocks commonly reimplement higher-level code. Assembly usage is correct. Most blocks have a NatSpec or inline documentation specifying what the code is doing, how it is implemented, and what the higher-level equivalence would be. Stateless fuzz tests showed that both results match for several functions. In other cases, where the code is borrowed from other projects, this is documented, and links to the original	**Satisfactory**

	repositories are provided. Some arithmetic operations are also fully implemented in assembly.	
Testing and Verification	A total of 590 tests were provided. The test suite runs “out of the box,” and there are no failing tests. However, test coverage is not 100% for some files and functions. Contracts and libraries have unitary and basic fuzzing tests. The tests run in the CI/CD pipeline for new pull requests and merge operations. Additionally, forge test coverage is integrated in the CI and displayed in each PR.	**Satisfactory**
Transaction Ordering	The lock/unlock mechanism makes all actions inside a transaction atomic, minimizing the risks of front-running and malicious MEV actors. The only transaction ordering risk present in the codebase is considered out of scope, as it was already reported and consists of front-running the pool initialization transaction.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Strict equality on fee comparison can cause fees to exceed 100%	Data Validation	**Informational**
2	Incorrect variable usage on swap fee	Data Validation	**Informational**
3	Collected protocol fees may count against user's currency deltas	Undefined Behavior	**Low**
4	Use of incorrect mask to clear higher bits of the protocolFee value	Data Validation	**Informational**
5	Insufficient event generation	Auditing and Logging	**Informational**
6	Similar-looking pool IDs can be brute-forced through the PoolKey hooks fields	Undefined Behavior	**Informational**

Detailed Findings

1. Strict equality on fee comparison can cause fees to exceed 100%	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-UNI4-1
Target: `libraries/Pool.sol`	

Description
The usage of strict equality on the max fee validation can lead to acceptance of an incorrect fee.

When performing a swap, the fee comprises a protocol and an LP fee, and is calculated through the `calculateSwapFee` function:

```
swapFee = protocolFee == 0 ? lpFee : uint16(protocolFee).calculateSwapFee(lpFee);
```

Figure 1.1: src/libraries/Pool.sol#L307

`swapFee` is represented as a percentage. It is checked to not be equal to 100% (`MAX_LP_FEE`):

```
if (swapFee == LPFeeLibrary.MAX_LP_FEE && !exactInput) {
    InvalidFeeForExactOut.selector.revertWith();
}
```

Figure 1.2: src/libraries/Pool.sol#L312-L314

Due to the usage of a strict equality (==), if the fee exceeds 100%, the validation passes, causing the fee to be greater than expected.

Note that the issue is not currently exploitable, as:

- We could not find a realistic way to increase the fee above 100%.
- The following operations in `computeSwapStep` would revert (e.g.,: `MAX_FEE_PIPS - _feePips`).

This issue's severity can be higher if combined with TOB-UNI4-2.

Recommendations

Short term, use >= instead of == when comparing the swap fee against its max value.

Long term, create tests for which the different fee limits are not set to 100%.

2. Incorrect variable usage on swap fee

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-UNI4-2
Target: `libraries/Pool.sol`	

Description
The swap fee is compared against the max LP fee constant instead of the max swap fee constant.

The swap fee is composed of two components: the protocol fee and the LP fee. swapFee is represented as a percentage. It is checked to not be equal to 100%:

```
if (swapFee == LPFeeLibrary.MAX_LP_FEE && !exactInput) {
    InvalidFeeForExactOut.selector.revertWith();
}
```

Figure 2.1: src/libraries/Pool.sol#L312-L314

However, the variable used for the comparison is the max LP fee (`MAX_LP_FEE`) instead of the max swap fee (`MAX_FEE_PIPS`):

```
/// @notice the lp fee is represented in hundredths of a bip, so the max is 100%
uint24 public constant MAX_LP_FEE = 1000000;
```

Figure 2.2: src/libraries/LPFeeLibrary.sol#L24-L25

```
library SwapMath {
    uint256 internal constant MAX_FEE_PIPS = 1e6;
```

Figure 2.3: src/libraries/SwapMath.sol#L9-L10

Both constants have the same value (10**6), so this issue is not an immediate threat to the protocol. However, this issue's severity would be higher if combined with TOB-UNI4-1.

Exploit scenario
The LP fee is updated to be at maximum 10%, and the protocol fee is expected to be 5%. As the swap fee is capped at the LP fee amount (10%), the swap fee is incorrect.

Recommendations
Short term, use `SwapMath.MAX_FEE_PIPS` instead of `LPFeeLibrary.MAX_LP_FEE` when comparing the swap fee against its max value.

Long term, create tests for which the different fee limits are not set to 100%.

3. Collected protocol fees may count against user's currency deltas

Severity: **Low**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-UNI4-3
Target: `PoolManager.sol`	

Description

Uniswap v4's protocol-level fee collection operates outside of the `currencyDelta` model used by the rest of the protocol. This creates an opportunity for erroneous `settle` calculations if protocol fee collection is performed after `sync` is called, ultimately leading to an unexpected revert.

When a user conducts a swap, position adjustment, or other action, it generates `currencyDeltas` for that user to represent the amount owed to the user or owed to the protocol. These `currencyDeltas` are cleared by calling `sync(currency)`, paying the amount of debt owed, then calling `settle`.

The `sync` and `settle` functions determine how much the user has paid by comparing the difference between `currency.balanceOfSelf` when both `sync` and `settle` are called. If the user calls `collectProtocolFees` between `sync` and `settle`, the amount of fees paid to the recipient will erroneously count against the user's `currencyDelta`, as if the user had called `take` or `mint` for the amount of fees paid.

Exploit scenario

The Uniswap DAO votes to turn on the protocol fee switch, and creates a contract that will harvest protocol fees and then swap them for the Uniswap Protocol governance token. The contract erroneously calls the `collectProtocolFees` function between `sync` and `settle`, and when determining how much needs to be paid to successfully settle the transaction, it manually calculates the `currencyDelta`.

In this situation, the fee collection process either: 1. becomes a no-op that collects fees and burns them by sending them to the v4 singleton, or 2. the transaction reverts.

Recommendations

Short term, add a guard to the `collectProtocolFees()` function to prevent it from being called while the contract is unlocked, and add a guard to `sync` to ensure it can only be called when the singleton is unlocked. Alternatively, add comments or documentation regarding the safe use of `collectPoolFees`.

Long term, add stateful properties to detect this kind of balance tampering attack in the future.

4. Use of incorrect mask to clear higher bits of the protocolFee value

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-UNI4-4
Target: `libraries/ProtocolFeeLibrary.sol`	

Description

The `calculateSwapFee` function of the `ProtocolFeeLibrary` contract uses an incorrect mask of `0xffff` to clear higher bits of the `protocolFee` value, which is a 12-bit value.

The `swap` function of the `Pool` library contract loads the protocol fee from the storage variable `slot0` of the singleton contract and calls one of the `getZeroForOneFee` or `getOneForZeroFee` functions to obtain the protocol fee percentage value:

```
uint256 protocolFee =
    zeroForOne ? slot0Start.protocolFee().getZeroForOneFee() :
slot0Start.protocolFee().getOneForZeroFee();
```

Figure 4.1: `libraries/Pool.sol#L291-L292`

The `getZeroForOneFee` function of the `ProtocolFeeLibrary` contract captures the lower 12 bits of the storage value and returns them in a `uint16` type value:

```
function getZeroForOneFee(uint24 self) internal pure returns (uint16) {
    return uint16(self & 0xfff);
}
```

Figure 4.2: `libraries/ProtocolFeeLibrary.sol#L17-L19`

Next, the `swap` function of the `Pool` library contract calls the `calculateSwapFee` function on the `protocolFee` variable of type `uint16` to compute the swap fee amount, combining the protocol fee and liquidity provider fee. However, the `calculateSwapFee` function assumes the value of the `self` variable, which is the `protocolFee` variable, to be of 16-bit length instead of 12-bit length and uses a mask of `0xffff` instead of `0xfff` to clear higher bits of the provided value:

```
function calculateSwapFee(uint16 self, uint24 lpFee) internal pure returns (uint24
swapFee) {
    // protocolFee + lpFee - (protocolFee * lpFee / 1_000_000). Div rounds up to
favor LPs over the protocol.
    assembly ("memory-safe") {
        self := and(self, 0xffff)
```

```
        lpFee := and(lpFee, 0xffffff)
        let numerator := mul(self, lpFee)
        let divRoundingUp := add(div(numerator, PIPS_DENOMINATOR), gt(mod(numerator,
PIPS_DENOMINATOR), 0))
        swapFee := sub(add(self, lpFee), divRoundingUp)
    }
}
```

Figure 4.3: libraries/ProtocolFeeLibrary.sol#L38-L47

Usage of an incorrect mask does not lead to incorrect calculations or financial loss in the current implementation because of correct masking in the `getZeroForOneFee` or `getOneForZeroFee` functions. It could lead to a higher fee being charged to the user if the `calculateSwapFee` function was called on a `uint16` value that did not have its upper four bits cleared.

Recommendations

Short term, use the correct mask `0xfff` to clear higher bits of the `protocolFee` value and document this behavior in inline code comments.

Long term, consider actual limits of values instead of the types when sanitizing the values for arithmetic operations.

5. Insufficient event generation	
Severity: **Informational**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-UNI4-5
Target: Various	

Description
Multiple critical operations do not emit events. As a result, it will be difficult to review the correct behavior of the contracts once they have been deployed.

Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions; malfunctioning contracts and attacks could go undetected.

The following operation should trigger events:

- `PoolManager`
  - `updateDynamicLPFee`

The following operation should modify events:

- `PoolManager`
  - The `initialize` function should include `sqrtPriceX96` as an event parameter if it emits.

The following operations may also be considered to trigger events. If they do not, they should be documented properly.

- `PoolManager`
  - `donate`
  - `settle`
  - `take`

Recommendations
Short term, add events for all operations that could contribute to a higher level of monitoring and alerting. If certain operations are not set up to emit events to optimize gas usage, they should be comprehensively documented.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A

monitoring mechanism for critical events would quickly detect any compromised system components.

6. Similar-looking pool IDs can be brute-forced through the PoolKey hooks fields

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-UNI4-6
Target: `types/PoolId.sol`, `types/PoolKey.sol`	

Description

Similar-looking pool IDs can be brute-forced by using the `PoolKey.hooks` field as a nonce. Attackers could use this to trick users into using their malicious pools. This does not affect the Uniswap v4 protocol; however, this does impact third-party integrators, which should try to minimize users falling victim to using malicious pools.

Pools can be freely created in Uniswap v4 through the `PoolManager.initialize` function. Each Pool has five fields (see figure 6.1), and the hash of these results in the pool's ID (see figure 6.2).

```
 8    struct PoolKey {
 9        /// @notice The lower currency of the pool, sorted numerically
10        Currency currency0;
11        /// @notice The higher currency of the pool, sorted numerically
12        Currency currency1;
13        /// @notice The pool swap fee, capped at 1_000_000. If the highest bit is
1, the pool has a dynamic fee and must be exactly equal to 0x800000
14        uint24 fee;
15        /// @notice Ticks that involve positions must be a multiple of tick
spacing
16        int24 tickSpacing;
17        /// @notice The hooks of the pool
18        IHooks hooks;
19    }
```

Figure 6.1: The `PoolKey` struct in `types/PoolKey.sol#L8-L19`

```
 9    library PoolIdLibrary {
10        /// @notice Returns value equal to keccak256(abi.encode(poolKey))
11        function toId(PoolKey memory poolKey) internal pure returns (PoolId
poolId) {
12            assembly ("memory-safe") {
13                poolId := keccak256(poolKey, mul(32, 5))
14            }
15        }
16    }
```

Figure 6.2: Hashing of the PoolKey struct in types/PoolId.sol#L9-L16

An attacker can create a pool with a malicious Hooks contract that looks similar to a legitimate victim pool by fulfilling the following requirements:

1. **The currency0, currency1, fee, and tickSpacing fields are identical to the victim pool.** This can be easily achieved since multiple pools with the same currency0, currency1, fee, and tickSpacing fields can exist as long as they have a different hooks field (which happens to be one of the attacker's other requirements; see below).
2. **The ID of the malicious pool is similar to the ID of the victim pool (i.e., the first and last X characters are identical).** This can be achieved by treating the hooks field as a nonce and brute-forcing it until a desired keccak256 hash (= pool ID) is generated. UIs tend to shorten long strings of hexadecimal characters, in which case having an identical first and last X character could be enough to trick users.
3. **The attacker's custom Hooks contract is deployed at the generated hooks address.** This can be achieved by using a CREATE2 (or CREATE3) factory to precompute a deployment address that, when placed in the PoolKey.hooks field, generates a pool ID that is similar to the victim pool's ID.
4. **The address of the Hooks contract has the right flags enabled (present in the lower 14 bits).** This can be achieved by requiring that the CREATE2 (or CREATE3) precomputed address has the lower 14 bits set as desired to enable the right flags.

In Uniswap v3, this problem did not exist since each pool was deployed separately, there was no field that could be used as a nonce to generate a desired address, and only one token pair per fee tier (of which there are only a handful) could be created. In Uniswap v4, these limitations do not exist.

This begs the question: *how do web/mobile apps display pools to users so that they know which is "the right one"*? One solution is, of course, by showing the liquidity in a pool, which should immediately make clear which pool is the right pool to use (i.e., due to a large amount of liquidity). However, for newly created pools, there may not be a lot of liquidity upon (or within a short timeframe after) pool creation. Also, a well-funded attacker might actually create a pool and add a substantial amount of liquidity, which then later is withdrawn again.

Recommendations

Short term, there is no silver bullet solution to this problem. We therefore recommend documenting this issue in the official Uniswap v4 docs so that users and integrators are made aware.

Long term:

- **Design a checklist (or flow chart) for users that helps them to determine if a pool is likely safe to interact with.** This will not be easy since simply saying, "do not interact with a pool with less than 5 LPs" may actually filter out pools that are not malicious. The same goes for "do not interact with pools created less than 7 days ago," which filters out new projects that are not malicious. The best solution could be creating a list of things to check that add up to a certain score, indicating that the pool is probably not safe to interact with.
- **Design guidelines for integrators that explain what information to display in the UI for each pool.** These guidelines should be aligned to the checklist from the above point so that users can easily follow the checklist.
- **Consider creating a allowlist of pool IDs that are known to be non-malicious, and only display these by default in the Uniswap v4 UI.**
- **Consider creating a allowlist of hook contract addresses and/or hashes of runtime bytecode of hook contracts.** This could be used to show to users which hooks are known not to be malicious (although, due to the arbitrary nature of hooks contracts, they may still perform external calls that lead to malicious behavior). The flexibility of hooks contracts is both a plus—it provides flexibility for projects to build custom integrations—and a negative, due to all the ways hooks contracts can act maliciously.
- **Consider using a different way to determine a pool ID.** For example, an incrementing integer as ID would prevent this issue entirely. However, this would incur more gas costs since then the `PoolKey` struct field would need to be stored within the contract storage (and SLOADed during every action). And since lower gas costs is one of the design goals of Uniswap v4, this is probably not a valid solution. However, there may be other creative ways to generate a pool ID that we did not think of and that would prevent this issue.

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
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
Transaction Ordering	The system's resistance to transaction-ordering attacks

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. Code Quality Findings

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability. We recommend fixing the issues reported here.

- **Use two-step ownership transfer in `ProtocolFees`.** The `ProtocolFees` contract inherits from solmate's `Owned` contract, which features a single-step ownership transfer. The effect is that it is possible to lose access to the ProtocolFeeController setter. Given that this function is not meant to be called often, having an additional step for ownership transfer does not affect the runtime gas optimizations. Additionally, we recommend that the initial owner be different from the deployer.
- **Rename the `state` variable in `Pool.modifyLiquidity`.** The `Pool` library defines a `State` structure that contains the pool state. Later, in the same file, the `modifyLiquidity` function defines a local variable called `state`, of type `ModifyLiquidityState`. Given that the first argument to the function is a `State` structure, the similarity in names between the `State` structure and `state` variable makes the code difficult to read.
- **Fix the hashed value for the `NonZeroDeltaCount` slot.** The library name is `NonZeroDeltaCount`, and the value used to calculate the slot hash is NonzeroDeltaCount.
- **Fix incorrect comments in `Currency` data type.** The comments in the transfer function of the `Currency` library are incorrect.
- **Upgrade OpenZeppelin contracts to the latest version.** The OpenZeppelin contracts submodule uses version 4.4.2 of the contracts, while the latest release is version 5.0.2.
- **Fix the documentation for the `Hooks` library.** The official documentation website for hook deployment, mentions that the higher-order bits of the hook address are used as flags. However, the `Hooks` library code uses the lower-order bits.
- **Include zero-value checks for certain function arguments**. Certain function parameters do not contain zero-value checks, leading to token loss. More specifically, the function argument `to` in the `take` function, and the function argument `recipient` in `collectProtocolFees`, should be checked for the zero address.

D. Invariant Testing and Harness Design

When reviewing protocols with a large potential state space, Trail of Bits creates various stateful and stateless fuzz testing harnesses to verify system properties that would be challenging or even impossible to verify using manual review.

We performed automated testing using a combination of stateful fuzz testing harnesses run using Echidna and Medusa, and a set of stateless invariants tested using both fuzzing and formal methods.

Stateless Invariant Testing

As a general rule, we recommend favoring fuzzing over formal methods (see Why fuzzing over formal verification?). However, given the critical and low complexity of some invariants, we decided to use Halmos to check for their correctness. These invariants did not require any special harness and are documented in the Automated Testing section.

Stateful Invariant Testing

Stateful invariant testing involves initializing a system, then using a fuzzer to run various public functions with specially selected parameters in order to break user-defined invariants. The fuzzer runs up to `N` selected functions for a given EVM context before resetting the context and starting over. Stateful invariant testing is "stateful" because the EVM context is reused for the next transaction in the sequence, meaning each invariant may be run against exotic system states produced by the previous transactions.

Stateful invariant testing, while it does use a fuzzer, should not be confused with parameterized fuzzing provided by tools like `foundry fuzz`. Parameterized fuzzing allows invariants to be tested only against a very specific subset of the state space—specifically, whatever state the system is in after `setUp` is completed. This means that the amount of state space that can be verified by a parameterized fuzzing test is astronomically smaller than the state space that can be verified by a stateful invariant test.

For stateful invariant testing, we use two fuzzers, Echidna and Medusa, both maintained by Trail of Bits. These fuzzers are used in conjunction with a test harness: a special contract that sits in front of the system under test, and is called by the fuzzers to produce transaction sequences that explore the state space, as measured by code coverage.

Since there is no canonical way of performing stateful invariant tests, designing a test harness is more of an art than a science. In the following sections, we provide details about each harness we wrote, the design decisions that went into each harness, and the kinds of properties we expect to verify with each harness design.

Stateful Invariants Using the End-to-End Harness

At the beginning of the engagement, we started with a targeted end-to-end harness that could be used to verify the properties of specific user flows, such as swapping, providing liquidity, and donating. The goal of this harness is not only to test those specific flows, but also to help acquaint us with v4's new lock/unlock model, its existing test suite, and to determine whether parts of the existing unit test suite can be repurposed into a more generalized fuzzing harness.

Figure C.1 shows a harness diagram. The harness revolves around the `End2End` contract, which is used to initialize the harness, to deploy actor/ancillary contracts, and as an entrypoint for the fuzzer to trigger actions within the system. We borrowed much of the initialization and callback logic from Uniswap's `Deployer`, `PoolSwapTest`, `PoolDonateTest`, and similar contracts, drastically reducing the amount of time required to set up the harness.

The end-to-end harness may be initialized with a specified number of currencies, a specified number of distinct actors, and a dynamic number of pools that are initialized using fuzzed values. The full supply of each currency is minted to the end-to-end contract, and actors can borrow funds from the contract as needed for their tests. This allows actors to use funds up to the `totalSupply` for each currency, where if the values for each currency were distributed to each actor, each actor would only be able to test up to `totalSupply/numActors` tokens.

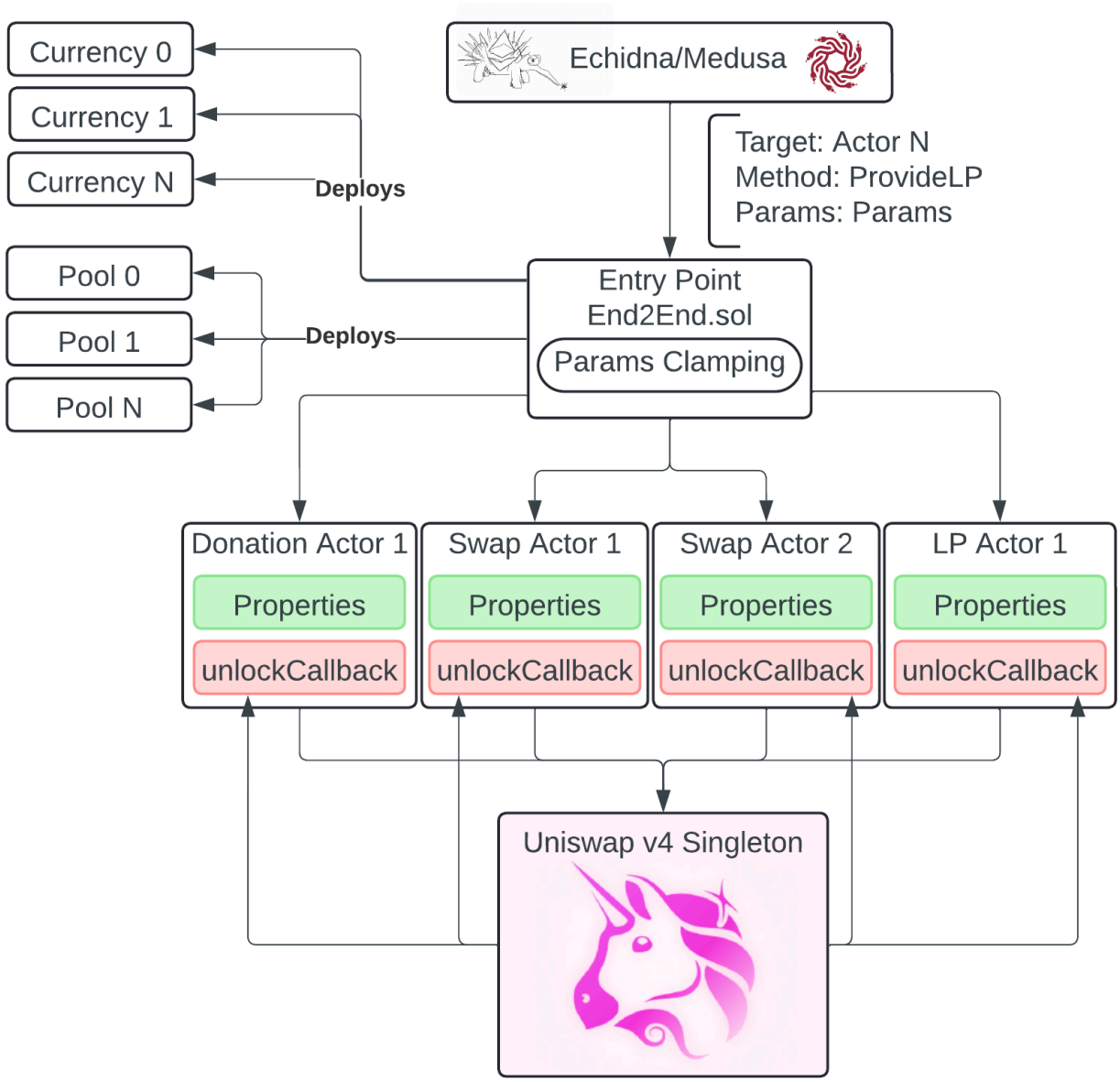

Figure C.1: Diagram of the end-to-end fuzzing harness

Stateful Invariants Using the Actions Harness

Shortly after completing the end-to-end harness, we noted severe limitations on the kinds of properties that could be verified by an end-to-end harness. Uniswap v4 lacks the atomic properties that its priors and most other smart contract protocols have, meaning the fuzzer cannot simply use `swap` or `modifyPosition` as entrypoints—they must be orchestrated with multiple other actions to produce a valid transaction. The end-to-end harness, by its nature, automatically performs this overhead action orchestration, thus preventing action-level properties from being directly verified.

This detail necessitated a drastically different harness design—one that could verify Uniswap v4's non-atomic "actions" in addition to its end-to-end properties.

We began designing a new harness that could address the following requirements:

1. The ability to test the state transition/side effects of each individual Uniswap v4 action (swap, `modifyPosition`, donate, settle, etc.)
2. The ability to test multiple unlock/lock contexts during the same transaction to verify that failing to clean transient storage does not cause issues.
3. The ability to detect unexpected overflows/underflows in v4's new arithmetic.
4. The ability to implement lock-wide properties, including properties defined in the previous end-to-end harness.
5. The ability to address shortcomings in the system's previous formal analysis—specifically the assumption that all loops iterate at most once.
6. Allow as many values to be parameterized as possible (e.g., variable number of currencies, variable number of pools, variable pool configuration).

Actions Harness Design

The core principle of the harness is to use the fuzzer to generate random sequences of actions with parameters, using one transaction for each action to be added to the sequence. Each action and its parameters are stored in the `actions` and `params` lists of `ActionFuzzEntrypoint`. One of the functions exposed to the fuzzer is `runActions`, which takes the actions and params lists and calls into `ActionsRouter.executeActions` to execute all of the actions in the same transaction.

`ActionsRouter` is a pre-existing contract used by the v4 test suite to run arbitrary lists of actions against the v4 singleton. When `ActionsRouter.executeActions(actions,params)` is called, it locks the v4 singleton, and then in its `unlockCallback()`, it sequentially runs each action.

We modified `ActionsRouter` to add a `HARNESS_CALLBACK` action. This action calls back into `ActionFuzzEntrypoint` instead of calling the v4 singleton, allowing our test harness to observe the state of the v4 singleton before it is called with an action, and compare it to the singleton's state after the call.

Figure C.2 shows a sequence diagram for the Actions Harness, and figure C.3 shows a block diagram of the harness.

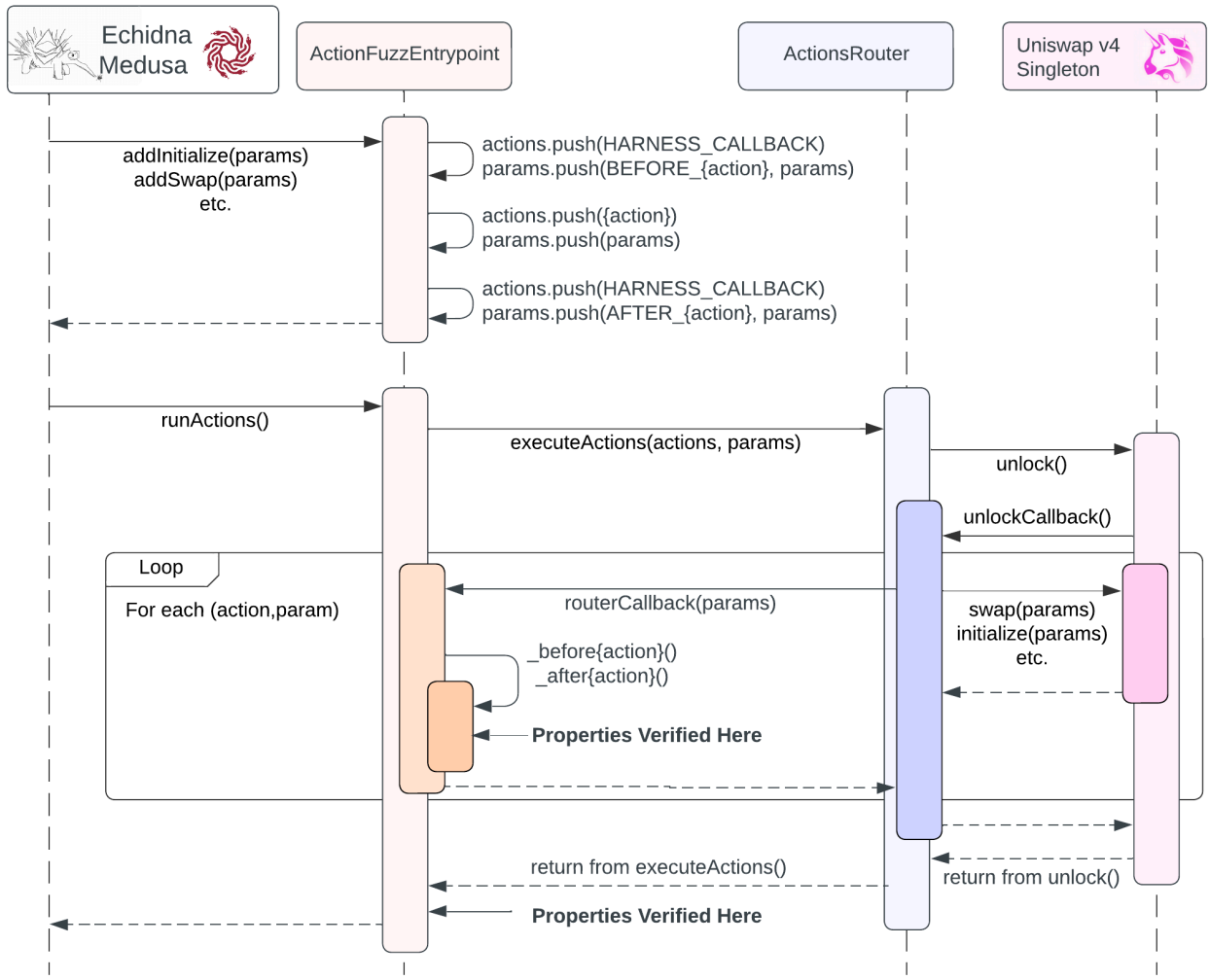

Figure C.2: Sequence diagram of the Actions fuzzing harness

Note that in figure C.2, that there are technically three touchpoints where the system's properties can be verified:

1. Before an action is called, we can verify that the system was left in a valid state. We can also capture the system's state before the action is called.

2. After an action is called, we can verify its side effects and ensure that the values returned by the action are correct.
3. After the system is relocked, we can verify unlock-wide, end-to-end properties. If multiple unlock/lock cycles occur during a given `runActions` transaction, we can verify properties relating to transient storage reuse.

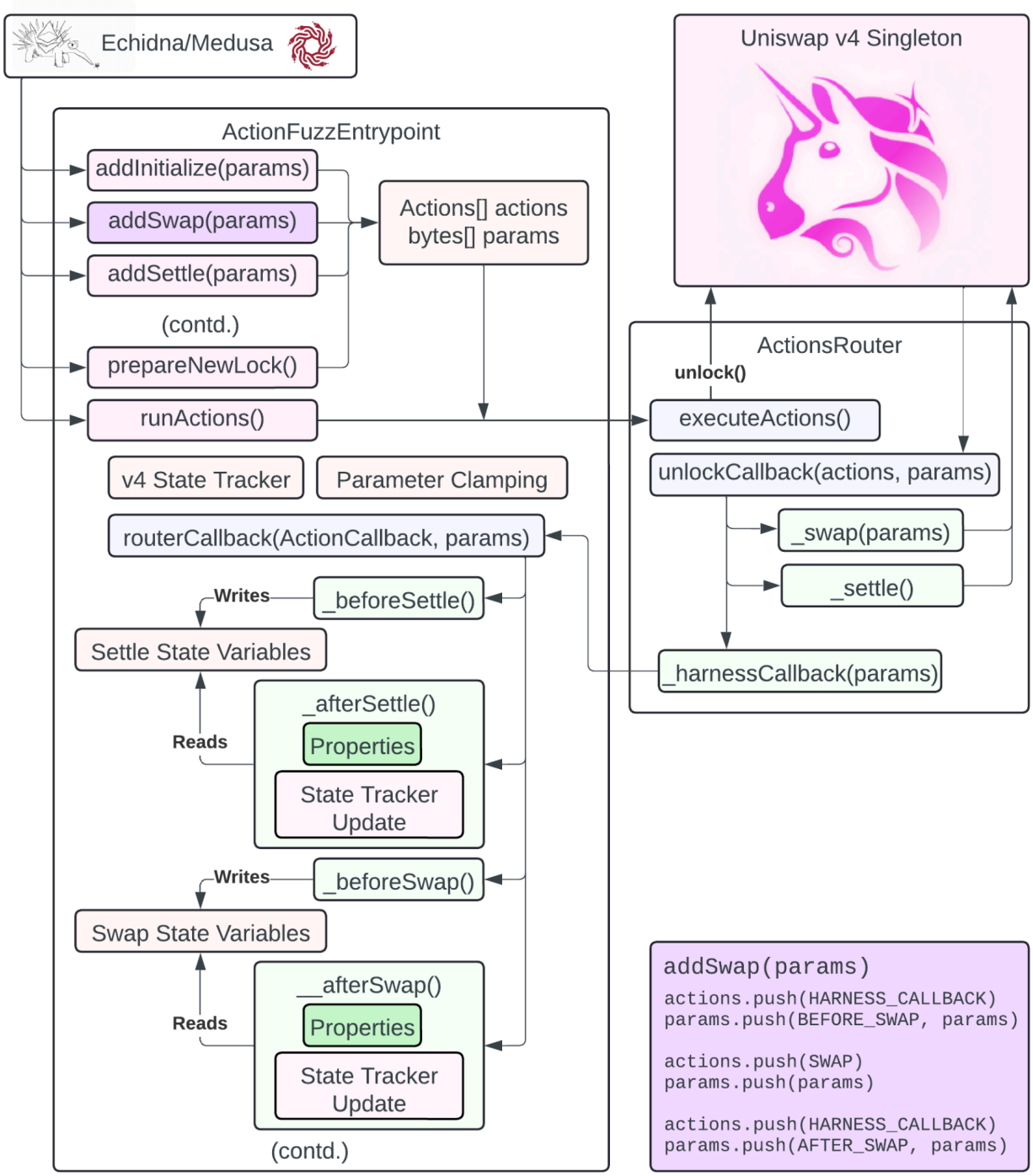

Figure C.3: Block diagram of the Actions fuzzing harness, including a characterization of addSwap's behavior

Selected Invariants for Discussion

At the beginning of the engagement, the client flagged several potential issues for which no known exploit existed, but there was also no proof that an exploit did not exist. The primary issue is #60 - Investigate Overflow Safety of Donate and feeGrowthGlobal.

The fundamental concern behind this issue is that it may be possible to overflow or underflow certain functions/variables in the system, causing one of the following effects:

1. An LP position's liquidity or fees become worth more than they should be.
2. A user may artificially increase or decrease their `currencyDelta` outside the intended functionality.

Since Uniswap v4 is implemented so that all pool balances are stored on the same contract, either of these issues would be of critical severity, since they would allow the malicious actor to steal liquidity belonging to another pool.

We decided to tackle this issue using two sets of properties: one set that ensures a pool's `FeeGrowthGlobal` cannot underflow, and one set that ensures there are always enough tokens in a given pool to pay out owed debts to liquidity providers and protocol fees.

Ensuring that a Pool's FeeGrowthGlobal Cannot Underflow

We implemented six properties to ensure that a pool's `FeeGrowthGlobal` does not underflow through the `donate` function; two to verify that the call's `BalanceDelta` is always zero or negative; two to verify that the fee growth matches the expected values based on the change in `BalanceDelta`; and two to verify that the `BalanceDelta`s returned by `donate` match the `amount0` and `amount1` that `donate` was called with (UNI-DONATE-1, UNI-DONATE-2, UNI-DONATE-6, UNI-DONATE-7, UNI-DONATE-8, UNI-DONATE-9). An abridged version of the code is shown in figure C.4.

Ensuring that the Singleton Can Always Cover Its Debts

At any one point, the debt owed by the singleton contract to creditors can be broken down into four categories:

1. Balances owed to creditors through their `currencyDelta`
2. Balances owed to the DAO through protocol fees
3. Accrued LP fees owed to liquidity providers
4. Liquidity owed to liquidity providers

After each action is executed, we can check these properties to ensure that they hold, using `_verifyGlobalProperties`. This function contains properties UNI-ACTION-1, UNI-ACTION-2, and UNI-ACTION-3, which ensure that the sum of all debts and credits does not exceed the singleton's balance.

There are some additional properties related to individual categories of debt, such as UNI-MODLIQ-8 and UNI-MODLIQ-9, which verify the amount of liquidity being withdrawn by a liquidity provider does not exceed the amount of liquidity available for the pool. This effectively verifies that when withdrawing liquidity, that amount will not be "taken" from another pool.

These are accompanied by UNI-MODLIQ-6 and UNI-MODLIQ-7, which verify that the singleton has adequate balance to credit the liquidity provider for their accrued fees. These two properties make more sense by scoping them down from the singleton level to the pool level, but due to time constraints, we were unable to complete this.

```
function _afterDonate(BalanceDelta delta) internal {
    // UNI-DONATE-6
    assertLte(delta.amount0(), 0, "A donate() call must not return a positive
BalanceDelta for currency0");
    // UNI-DONATE-8
    assertEq(_donateAmount0, -delta.amount0, "The donate() call BalanceDelta must
match the amount donated for amount0");
    [...]
    // how far until the 128x128 overflows
    uint256 growth0OverheadX128 = type(uint256).max -
_feeGrowthGlobalBeforeDonate0X128;

    // the expected change in fee growth based on delta
    uint256 feeGrowthDelta0X128 =
FullMath.mulDiv(uint256(uint128(-delta.amount0())), FixedPoint128.Q128, liquidity);

    uint256 feeGrowth0ExpectedX128 = _calculateExpectedFeeDelta(
        feeGrowthDelta0X128,
        _feeGrowthGlobalBeforeDonate0X128
    );
    (uint256 feeGrowth0AfterX128, uint256 feeGrowth1AfterX128) =
manager.getFeeGrowthGlobals(donatePoolId);
    [...]
    if (liquidity > 0 ) {
        if(_donateAmount0 > 0) {
            // UNI-DONATE-1
            assertEq(feeGrowth0ExpectedX128, feeGrowth0AfterX128 , "After a donation
with a non-zero amount0, the pool's feeGrowthGlobal0X128 be equal to the amount0
BalanceDelta, accounting for overflows.");
```

Figure C.4: An abridged reproduction of the donation properties verified in `_afterDonate`. (`audit-uniswap-v4/test/trailofbits/actionprops/DonateActionProps.sol`)

Future Work

There are several areas for potential future work to further improve the invariant test suite that we could not accomplish during this engagement due to time constraints.

1. Some critical properties, such as UNI-MODLIQ-6 and UNI-MODLIQ-7, are over-generalized, and would greatly benefit from being tightened up.
2. Some properties were not implemented due to time constraints, such as calculations for the change in `feeGrowthGlobal` resulting from a `swap` call.
3. Many properties of `modifyLiquidity` and `settleNative` remain incomplete, and are tested only at discrete points in the state space using the unit test suite.
4. Properties that verify that a function call *should not* revert. Due to time constraints, only several of these could be implemented, and only for the `initialize()` function.
5. Some potential properties could be added to better verify that the singleton's debts never exceed its assets by verifying the levels of debt of each individual pool instead of those in the singleton as a whole. Adding properties that operate at the pool-level will provide higher assurance and will be easier for the fuzzer to discover. UNI-MODLIQ-8 and UNI-MODLIQ-9 provide some examples for what this would look like, and it should be straightforward to expand to other forms of debt.

E. Static Invariants

Throughout the review, we identified multiple code patterns that require to be enforced through the codebase. To ensure their correctness, we wrote a linter tool based on slither.

The linter checks the following:

ID	Property	Why	Result
`noSelfCall_should_not_return`	Functions that use `noSelfCall` do not return any variable	The modifier is a no-op; any return variable would always be its default value.	**Passed**
`callHook`	Functions that calls `callHook` are protected against self-calls	This ensures that the hook will not re-enter to itself.	**Passed**
`pool_manager_function_ids`	`PoolManager`'s functions do not collide with the `Hooks` function	A function collision could lead to setting a hook to be the pool manager itself and executing unexpected code (e.g., having the manager swap assets).	**Passed**
`pool_manager_payable`	Only `settle`/`settleFor` are payable in the pool manager	Only these two functions should receive funds.	**Passed**

The code of the linter is provided below. We recommend that Uniswap add the tool in the CI, and extend it with further analysis:

```
from collections import defaultdict
from slither import Slither
from slither.core.declarations import Function
from slither.slithir.operations import TypeConversion, Binary, BinaryType
from slither.core.solidity_types.elementary_type import ElementaryType
from slither.core.declarations.solidity_variables import SolidityVariableComposed
from slither.utils.function import get_function_id


def noSelfCall_should_not_return(sl: Slither):
    # Check that the function that use noSelfCall don't return variable
    # Given than the modifier is a no-op, that would lead to function to return
```

```
default values

    no_finding_or_error = True
    hook_contracts = sl.get_contract_from_name("Hooks")

    for hook_contract in hook_contracts:
        noSelfCall = None
        for modifier in hook_contract.modifiers:
            if modifier.full_name == "noSelfCall(IHooks)":
                noSelfCall = modifier

        if not noSelfCall:
            print(f"noSelfCall not found in {hook_contract}")
            no_finding_or_error = False

        for function in hook_contract.functions:
            if noSelfCall in function.modifiers and function.returns:
                print(f"{function} has the {noSelfCall} modifier and return
variables")
                no_finding_or_error = False

    if no_finding_or_error:
        print(f" - [X] noSelfCall_should_not_return analyzed (no finding)")
    else:
        print(
            f" - [ ] noSelfCall_should_not_return analyzed (incomplete or with
finding)"
        )


def _has_msg_sender_self_check(function: Function) -> bool:

    self = []
    for ir in function.slithir_operations:
        if (
            isinstance(ir, TypeConversion)
            and ir.variable.name == "self"
            and ir.type == ElementaryType("address")
        ):
            self.append(ir.lvalue)

    for ir in function.slithir_operations:
        if isinstance(ir, Binary) and ir.type == BinaryType.EQUAL:
            if (
                ir.variable_left == SolidityVariableComposed("msg.sender")
                and ir.variable_right in self
            ):
                return True

    return False


def callHook(sl: Slither):
```

```
    # Check that the function that calls callHook have either
    # - The noSelfCall modifier
    # - Or compare msg.sender with self

    no_finding_or_error = True
    hook_contracts = sl.get_contract_from_name("Hooks")

    for hook_contract in hook_contracts:

        callHook =
hook_contract.get_function_from_signature("callHook(address,bytes)")

        noSelfCall = None
        for modifier in hook_contract.modifiers:
            if modifier.full_name == "noSelfCall(IHooks)":
                noSelfCall = modifier

        if not callHook or not noSelfCall:
            print(f"callHook or noSelfCall not found in {hook_contract}")
            no_finding_or_error = False

        for function in hook_contract.functions:
            # Allowlist callHookWithReturnDelta
            if function.name in ["callHookWithReturnDelta"]:
                continue

            if callHook in function.all_internal_calls():
                if noSelfCall in function.modifiers:
                    continue
                if _has_msg_sender_self_check(function):
                    continue

                print(f"{function} has is missing noSelfCall or msg.sender==self
check")
                no_finding_or_error = False

    if no_finding_or_error:
        print(f" - [X] callHook analyzed (no finding)")
    else:
        print(
            f" - [ ] callHook analyzed (incomplete or with finding)"
        )


def pool_manager_function_ids(sl: Slither):
    # Check that all the public functions of PoolManager dont collide with the hooks
functions (func id)
    # This is to prevent a hook to point to the pool manager to executed unexpected
code
    # (ex: having the manager swapping assets)

    no_finding_or_error = True
    pool_manager_contracts = sl.get_contract_from_name("PoolManager")
```

```
    hooks = sl.get_contract_from_name("IHooks")

    entry_ids = defaultdict(set)
    hooks_ids = defaultdict(set)
    for pool_manager_contract in pool_manager_contracts:

        for entry_point in pool_manager_contract.functions_entry_points:
            entry_ids[get_function_id(entry_point.solidity_signature)].add(
                entry_point.solidity_signature
            )

    for hook_contract in hooks:

        for hook in hook_contract.functions_entry_points:
            hooks_ids[get_function_id(hook.solidity_signature)].add(
                hook.solidity_signature
            )

    inter = set(entry_ids.keys()).intersection(set(hooks_ids.keys()))
    for i in inter:
        print(f"ID collision between {entry_ids[i]} and {hooks_ids[i]}")
        no_finding_or_error = False

    if no_finding_or_error:
        print(f" - [X] pool_manager_function_ids analyzed (no finding)")
    else:
        print(f" - [ ] pool_manager_function_ids analyzed (with finding)")


def pool_manager_payable(sl: Slither):
    # Check that only settle/settleFor are payable in the pool manager

    no_finding_or_error = True
    pool_manager_contracts = sl.get_contract_from_name("PoolManager")

    for manager in pool_manager_contracts:
        for function in manager.functions:
            if function.payable and not function.name in ["settle", "settleFor"]:
                print(f"{function} should not be payable")
                no_finding_or_error = False

    if no_finding_or_error:
        print(f" - [X] pool_manager_payable analyzed (no finding)")
    else:
        print(f" - [ ] pool_manager_payable analyzed (with finding)")


def main() -> None:

    # Run with python linters/linter.py
    # If call from another path, update "." to point to the top level directory of
the project
    sl = Slither(".")
```

```
    noSelfCall_should_not_return(sl)
    callHook(sl)
    pool_manager_function_ids(sl)
    pool_manager_payable(sl)


if __name__ == "__main__":
    main()
```

Figure E.1: Slither script

F. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On August 30, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Uniswap team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the six issues described in this report, Uniswap has resolved four issues, partially resolved one issue, and decided to not resolve one issue. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Strict equality on fee comparison can lead the fees to be greater than 100%	Resolved
2	Incorrect variable usage on swap fee	Resolved
3	Collected protocol fees may count against user's currency deltas	Resolved
4	Use of incorrect mask to clear higher bits of the protocolFee value	Resolved
5	Insufficient event generation	Partially Resolved
6	Similar-looking pool IDs can be brute forced through the PoolKey hooks fields	Unresolved

Detailed Fix Review Results

TOB-UNI4-1: Strict equality on fee comparison can cause fees to exceed 100%
Resolved in PR 836. The comparison was updated from is-equal-to (==) to is-greater-than-or-equal-to (>=).

TOB-UNI4-2: Incorrect variable usage on swap fee
Resolved in PR 831. A new constant (`SwapMath.MAX_SWAP_FEE`) was added to the implementation. Introducing a new constant named `MAX_SWAP_FEE` is more explicit than using the `MAX_FEE_PIPS` constant, which we originally recommended. All places where the max swap fee is needed now use this new constant. In other words, the use of the

`LPFeeLibrary.MAX_LP_FEE` and `SwapMath.MAX_FEE_PIPS` constants was replaced by using the `SwapMath.MAX_SWAP_FEE` constant in all applicable locations.

TOB-UNI4-3: Collected protocol fees may count against user's currency deltas
Resolved in PR 856. A guard was added to the sync function to ensure it can only be called when the contract is in the unlocked state. Additionally, a check was added in the `collectProtocolFees` function to prevent it from being called when the contract is unlocked, throwing a custom error (`ContractUnlocked`) if this check fails. Additionally, new tests were added to specifically test for this edge case and to ensure that the updated implementation correctly handles it.

TOB-UNI4-4: Use of incorrect mask to clear higher bits of the protocolFee value
Resolved in PR 835. The implementation was updated to use a 12-bit mask instead of a 16-bit mask.

TOB-UNI4-5: Insufficient event generation
Partially resolved in PR 845 and PR 808. The implementation was updated to emit an event in the `donate` function, and the existing `Initialized` event was updated to include the `sqrtPriceX96` value. The Uniswap team decided against adding an event in the `updateDynamicLPFee` function since hooks can also return a dynamic LP fee amount, which would not necessarily be emitted in an event. To normalize the behavior across all changes of updated dynamic LP fees, no event will be emitted from the `updateDynamicLPFee` function.

TOB-UNI4-6: Similar-looking pool IDs can be brute-forced through the PoolKey hooks fields
Unresolved. The Uniswap team decided to not resolve this issue.

G. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.