Uniswap V2 Audit Report

dapp.org

fv@dapp.org.uk

Table of Contents

Summary

From January 8th to April 30th 2020, a team of six engineers reviewed and formally verified crucial components of the smart contracts for the second version of the Uniswap decentralized trading protocol.

This work was carried out against the following git repositories:

- `uniswap-v2-core` at commit `8160750`
- `uniswap-v2-periphery` during ongoing development

During this period, two medium severity issues were discovered:

- Pair: fix to liquidity deflation introduces race condition
- Router: incompatible with token with fees on transfer

and a number of small fixes and gas efficiency improvements were made.

The full list of findings can be found here.

All issues discovered in the `uniswap-v2-core` repository during the course of this engagement have been addressed in the commit `8160750`.

The team recommends more analysis and review to be performed for the final versions of the peripheral contracts before launch.

Scope

Formal verification of the core smart contracts

Formal specifications and proofs of method-level correctness for the smart contracts in the `uniswap-v2-core` repository (namely `UniswapV2Pair` and `UniswapV2Factory`) were produced using the `act` specification language and K framework.

An overview of the formal specifications and their proof status can be found in the audit teams CI system at dapp.ci/k-uniswap, and a discussion on the nature of the proofs produced can be found in the Formal Verification section of this report.

Code review of core smart contracts

The audit team carried out a general code review and security analysis of the smart contracts of `uniswap-v2-core`, finding 2 bugs:

- 1 medium severity issue Pair: fix to liquidity deflation introduces race condition
- 1 low severity issue Math: integer overflow in `sqrt`

along with several recommendations regarding gas optimisation and code clarity, which are presented in detail in the Findings.

Most of these improvements were adopted by the Uniswap development team.

Numerical error analysis

The audit team performed a review of the numerical error incurred during contract execution, with a focus on the desirable invariants proposed by the Uniswap development team (e.g. the rounding error, if any, in a swap, favours the pool).

The team did not find any locations where numeric errors violated these invariants, and additionally produced a novel manual proof of correctness for the Babylonian square root algorithm in the fixed precision setting.

Code review of periphery smart contracts (during ongoing development)

The audit team also committed to carry out a general code review and security analysis of the smart contracts in the `uniswap-v2-periphery` repository.

Another suite of tests, including property based fuzzing tests using the dapp testing framework were developed during this process and are available here. These tests are targeting a now outdated version of the periphery contracts.

This work was carried out on a best effort basis, and the rate of change and state of active development in the periphery means that the review was performed on a cursory level only.

The team recommends more analysis and review to be performed for the final versions of the peripheral contracts before launch.

One medium severity issue was identified in the Router:

- Router: incompatible with token with fees on transfer

Team

The team included the authors of klab, an interactive proof explorer and verification tool chain, and act, a literate formal specification language for smart contracts. The team was also responsible for smart contract development and formal verification at MakerDAO: work that culminated in the implementation and formal verification of multi collateral Dai.

- David Currin
- David Terry
- Denis Erfurt
- Lev Livnev
- Lorenzo Manacorda
- Martin Lundfall

Findings

Recommendation	Type	Severity	Likelihood	Accepted	Commit
Router: incompatible with token with fees on transfer	Bug	Medium	High	Yes	
Pair: fix to liquidity deflation introduces race condition	Bug	Medium	Medium	Yes	`uniswap-v2-core@cbe801b`
Math: integer	Bug	Low	Low	Yes	`uniswap-v2-`

Recommendation	Type	Severity	Likelihood	Accepted	Commit
overflow in `sqrt`					`core@d1c8612`
ERC20: make `name`, `decimals`, `symbol` constant	Improvement	-	-	Yes	`uniswap-v2-core@cbe801b`
ERC20: remove `forfeit`	Improvement	-	-	Yes	`uniswap-v2-core@cbe801b`
Factory: use `.creationCode` when retrieving Pair bytecode	Improvement	-	-	Yes	`uniswap-v2-core@f2d4021`
Pair: replace block height with timestamp	Improvement	-	-	Yes	`uniswap-v2-core@a55aa4b`
Factory: replace `allPairs` array with a counter	Improvement	-	-	No	
Meta: replace math libraries with an inherited contract	Improvement	-	-	No	
Pair: divide by zero in `burn`	Improvement	-	-	No	

Bugs

Router: incompatible with token with fees on transfer

The user facing contract UniswapV2Router01.sol calculates the amount a user needs to transfer to `UniswapV2Pair.sol` in order to perform a mint, swap or burn.

In cases dealing with a token which subtracts a fee on `transferFrom`, the amount received by `UniswapV2Pair` is smaller than the required amount to perform a successful swap, causing the whole call to fail when the fee adjusted invariant is checked at the end of `swap`.

This is only a problem with the router, and additional routers or token wrappers can be used to mitigate this issue.

Pair: fix to liquidity deflation introduces race condition

Dan Robinson discovered a vector by which an early liquidity provider can make it very costly for other liquidity providers to enter, allowing them to monopolize the liquidity pool:

1. Before there is any liquidity in the pool, the attacker sends a small amount of both tokens of the Pair, receiving a small amount of liquidity pool shares ("LP Tokens").
2. They then send a very large amount of the Pair tokens to the contract, and invokes `sync`.

This deflates the LP token to be worth a large amount of Pair tokens, which can increase the barrier to entry for other liquidity providers to the point where submitting enough tokens to yield 1 wei of LP tokens can cost millions of dollars.

The Uniswap team suggested a fix to this issue which would force the minimum supply of the LP tokens to be 10,000 units, imposing this constraint in both `mint` and `burn`. The reasoning was that this would ensure that the liquidity token would allow for enough granularity, even after someone attempted to increase the barrier to entry.

However, this fix introduced another problem, where a malicious liquidity provider could sacrifice 1 wei of LP tokens to effectively render the last 9,999 LP tokens of the contract irredeemable. In other words, the suggested fix could render liquidity providers unable to exit the contract and redeem their original liquidity and fees.

After discussions with the Uniswap team, another solution was finally settled on:

When providing the initial liquidity to the contract (i.e. when `totalSupply` is 0 in `mint`), the liquidity provider must sacrifice 1000 LP tokens (by sending them to `address(0)`).

This mitigates the issue in two ways:

- The granularity of the LP tokens is always at least 1000
- The cost of performing the LP inflation attack increases significantly, as the attacker is now not the sole holder of LP tokens.

This mitigation comes at the expense of introducing an additional cost for the initial liquidity provider, but this cost is expected to be low enough to be acceptable for most tokens. For tokens with a reasonable value per minimum denomination, 1000 wei does not have more than a modest economic value. Note however, that in the extreme case of a Pair for two tokens that are both worth around $1.00 per wei, the up-front cost for the initial liquidity provider will be around $2,000.

Math: integer overflow in `sqrt`

The `sqrt` function of the `Math` library contract implements the Babylonian method for calculating the integer square root of `y`, i.e. the largest integer `z` such that `z^2 <= y`.

Before uniswap-v2-core@d1c8612e, the initial value to the iteration was given as `x = (y + 1) / 2`, which results in an integer overflow when `y = uint(-1)`, setting `x` to zero, leading to a division by zero in the calculation of `x_(n+1)`.

This does not result in incorrect return value for `sqrt`, but does cause the function to revert unnecessarily. We suggested to change the initial value to `x = y / 2 + 1`, making `sqrt` well defined over its entire domain.

A manual proof of correctness of this algorithm is given here.

Improvements

ERC20: make `name`, `decimals`, `symbol` constant

The `name`, `decimals` and `symbol` attributes of the `UniswapV2ERC20` contract were set dynamically in the constructor. However, they were always set to the same value by the inheriting `UniswapV2Pair` contract, resulting in unnecessary bytecode complexity and gas consumption.

The audit team recommended that either each Pair should be given a unique name and symbol, or these attributes should be set as hard-coded `constant` in the `UniswapV2ERC20` contract.

Based on the following factors, the Uniswap team decided to make the attributes in question `constant`:

- String manipulation in Solidity is cumbersome and error-prone
- Uniswap V1 exchanges all have the same `name` and `symbol`, and no major issues have been observed in the wild

This change was implemented in `uniswap-v2-core@cbe801b5`.

ERC20: remove `forfeit`

The `forfeit` method of `UniswapV2ERC20` allowed anyone to burn their liquidity tokens without receiving tokens from the pool in return. It functioned effectively as a donation to all other members of the pool.

While in some situations such donations to the pool are desirable (e.g. Synthetix used token inflation rewards to incentivise liquidity providers in their Uniswap V1 exchanges), the same outcome can be achieved by transferring tokens to the Pair and calling `sync` to force the reserves to match the balances.

In the interests of minimizing the number of allowed state transitions in the core `UniswapV2Pair` contract, the audit team therefore recommended that the `forfeit` and `forfeitFrom` methods be removed.

The Uniswap team accepted this recommendation in `uniswap-v2-core@cbe801b5`.

Factory: use `.creationCode` when retrieving Pair bytecode

The `createPair` method of `UniswapV2Factory` stored the bytecode for a Pair as a raw `bytes` value set during construction. When deploying a new Pair, the bytecode would have to be loaded into memory from storage, word-by-word. The audit team recommended that this storage variable be removed, and replaced with the `.creationCode` attribute of the `UniswapV2Pair` contract, simplifying the code and clarifying intent.

Moving the bytecode out of storage also resulted in a significant gas saving (millions of gas) when creating Pairs, due to the reduction in the amount of expensive `SLOAD` operations required.

The Uniswap team accepted this recommendation in `uniswap-v2-core@f2d40214`.

Pair: replace block height with timestamp

`UniswapV2Pair` used the number of blocks elapsed between price updates in order to calculate block-weighted cumulative price. While there are valid arguments that block height is a more "objective" variable than the block timestamp, the audit team suggested that this be changed to use the time elapsed, measured by the change in block timestamp since the last price update.

For consumers that sample the Pair's price accumulators less frequently, e.g. oracles seeking to calculate time-weighted average prices over windows of weeks or months, the choice to use block-weighting or timestamp-weighting could result in material differences. Therefore, using block-weighting might create unexpected results for oracles targeting a time-weighted average (which, at least outside of the blockchain setting, is a more economically meaningful figure)

As a concrete example: the mean Ethereum block time has been known to spike by up to double that of its typical level in the weeks leading up to a "difficulty bomb" hard fork. A block-weighted price accumulator could cause an oracle price with a long averaging window to diverge substantially from the time-weighted average price, as computed by other, off-chain sources. To avoid unnecessarily exposing users to these implementation-related quirks, it may be better to use simple time-weighting.

Factory: replace `allPairs` array with a counter

The `allPairs` array in the Factory is primarily used to maintain a count of all created Pairs and the audit team noted that it could be replaced by a simple counter. The Uniswap team decided to keep the array implementation in case iteration over the array might be required. They were also advised that any code which iterates over the `allPairs` array should paginate, or otherwise break up the access into constant-gas chunks, and moreover that iterating over the array from an external contract would be very costly, requiring one call per element read.

Since Solidity uses the hash of the slot as the starting position for storage arrays, their maximum length before integer overflow of the array size depends on the slot number at which they are located: `maxUInt256 - keccak(uint256(slot)) + 1`

For the sake of completeness, and since it is an unusual property for overflow behaviour to depend on the order in which variables are declared in source code, we note that that Solidity provides no protection against the (negligible probability) overflow of `allPairs`.

Meta: replace math libraries with an inherited contract

The `UniswapV2Pair` contract depends on three libraries, `UQ112x112`, `SafeMath` and `Math`, providing additional `internal` functions and custom syntax for `uint256` safemath. Including these internal methods in the `UniswapV2Pair` directly (or inheriting them from another contract) instead makes creating new pairs about 15000 gas cheaper.

Since this change would make the use of custom syntax for `uint256` safemath impossible, the Uniswap team decided to not implement this suggestion.

Pair: divide by zero in `burn`

If `burn` is called when `totalSupply` is `0`, then the call will fail due to a divide by zero when calculating the amounts to be returned to the caller.

When a divide by zero is encountered, Solidity "throws" by executing the `INVALID` opcode, consuming all of the gas in the initiating call (unlike `REVERT`), resulting in an unnecessary monetary loss if `burn` is accidentally called before any LP shares have been minted. Additionally, this division by zero behaviour in Solidity might not be widely known, since it differs from other languages (like the EVM, in which `DIV` by `0` returns `0`).

Additionally, some analysis tools treat an INVALID opcode as an assertion violation, whereas here it is more akin to an unmet precondition. This can lead to false error reporting or otherwise complicate the analysis of the Uniswap contracts using these tools.

As the divide by zero occurs in such a limited set of circumstances, and has such a limited impact, the audit team presented it for informational purposes only and no change was made by the Uniswap team.

Design Comments

Front-Running and Transaction Reordering

An actor who can influence the order in which transactions are included in a block can affect the economic outcome of trades. The audit team is aware of two strategies for profitably exploiting this fact. Such strategies can be used not only by miners but also by any party who is able to observe unconfirmed transactions and submit their own transactions with carefully chosen gas prices. The router includes some features that provide some degree of protection against the first strategy, but no such mitigations exist for the second.

These issues are well known, and are also present in Uniswap V1. Transaction reordering and front-running attacks can arguably be viewed as a broader structural problem inherent to many trading venues with on-chain clearing and settlement. As has been extensively documented in Daian et al.[1], such strategies have been widespread in Ethereum for many years, and the rents accruing to miners from the transaction reordering privilege (miner extractable value) can even pose a threat to the security of the blockchain consensus layer.

Moving the market against the trader

In this variant, an actor who can observe unconfirmed transactions can attempt to insert trades before and after the target trade, manipulating the Pair's price in a way that will result in a profit for the front-runner, and a worse price for the trader.

As an example, consider a Pair for ETH/DAI. Suppose a trader sends a transaction to sell 1 ETH. If a front-runner is able to sandwich the trader's transaction between two trades which sell and buy ETH respectively, then it is clear that the state of the Pair when the trader's transaction is executed will result in a lower price of ETH for the trader, due to the front-runner's sale, and that the extent of the price deterioration will depend on the size of the front-runner's first trade (it can be shown by considering the constant product invariant that the relationship is quadratic). After the target trade has executed, the front-runner's second trade is executed, buying back a sufficient amount of ETH to return the Pair's price to roughly where it started.

By following the flow of balances, we observe that the aggregate economic effect of the sandwiching was equivalent to a trade between the front-runner and the trader (with the pool ending up back where they started, having only collected fees from the 3 transactions), however the front-runner was able to effectively choose the price at which the trade happens by choosing the size of their first trade, and so is able to set the price arbitrarily in their own favour. To see that this can still be profitable in the presence of LP fees, it suffices to observe that while LP fees paid by the front-runner are proportional to the size of the sandwich trade, the price impact of the sandwich trade on the Pair's price is quadratic in the size of the sandwich trade, meaning that the sandwich can be made profitable with a sufficiently large manipulating trade, and the trader's loss is only limited by the value of their trade.

A similar method can be used against a liquidity provider who is entering the pool.

The methods exposed on the user-facing `UniswapV2Router01` contract contain arguments that allow callers to impose off-chain slippage limits on their orders (`AmountOutMin`, `AmountOutMax`), and when set appropriately these parameters can limit the losses to trader front-running. Note however that it might not be possible to eliminate front-running entirely with this technique, since setting the slippage limit too tightly to the market could result in a poor success rate, with the market moving against the trader.

Sandwiching large trades with `mint` and `burn`

In this second variant, the attacker watches for large trades, and sandwiches the target trade with calls to `mint` and `burn` with a very large position relative to the initial size of the pool. The attacker is therefore able to extract a sizeable proportion of the LP fees for that trade without exposing themselves to the price risk inherent to providing liquidity on Uniswap.

Neither the core nor the periphery contracts contain guards against this attack, and the audit team is not aware of any straightforward solution. A minimum lock lock time for liquidity providers, imposed in the core, could potentially help to reduce the profits siphonable with this attack.

Liquidity providers should monitor the activity on Pairs in which they participate to evaluate their exposure to activity of this kind, since it could result in diminished returns.

Oracle Integrity

Even though AMM (automated market maker) systems such as the Uniswap protocol, by way of their automatic price discovery mechanism, seem to offer a compelling way forward for constructing a trust-minimised on-chain price oracle, this turns out to be non-trivial to implement safely. For instance, the naïve approach of simply querying the current price for a Pair from a smart contract turns out to be insecure in the majority of practical situations, since the cost of price manipulation is often low compared to the value that is at stake. To make matters worse, by synchronously executing a "de-manipulation" trade afterwards, the attacker can often recoup most of their manipulation costs: see samczsun's article[2] for a practical discussion of this class of oracle attacks.

As a result, constructing a safe price oracle on top of the V1 Uniswap protocol is highly non-trivial. For this reason, the V2 protocol introduces the time-weighted average price accumulators `price0CumulativeLast`, `price1CumulativeLast`. The accumulators track the Pair cumulative time-price at the end of each block. Sampling the accumulator at two points in time, taking the difference, and dividing by the elapsed time yields the time-weighted average price of the price in that Pair at the ends of the blocks during that time interval. From a robustness perspective, this differs from the instantaneous price in two crucial ways:

- the averaged price depends on prices that appeared in the past, proportionally to how long they appeared for, and the oracle consumer can choose the length of the period for averaging.
- the averaged price is not influenced by prices the appeared within a block,

but only by the final price at the end of a block. In particular, the average is not affected by prices arising during synchronous execution of multiple trades within a block.

The first point means that in order to manipulate an oracle which uses a longer averaging period, an attacker would need to maintain a manipulated price for a longer period of time. The second means that an attacker must maintain the manipulated price at the end of a block in order to have an effect on the average, which is expected to maximise the attacker's cost by reducing the likelihood that they will be the one to "de-manipulate" the price.

As a reference example, a contract called `ExampleOracleSimple` samples the accumulators for a Pair at most once per hour. Therefore, in order to manipulate the price provided by this oracle to be higher or lower than the true market price, an attacker would have to create a situation where the available price in a Uniswap Pair at the beginning of many blocks during a 1-hour period was significantly higher or lower than the market price.

Given that in most cases the attacker would not have a guarantee of recouping their costs by "de-manipulating", the costs of this manipulation could be roughly estimated by considering the required trade size to move the price to a given level (as a function of the pool size), and the number of blocks over the period. A possible starting point for such an analysis could be the calculations in Angeris et al.[3]. There are some technical and economic nuances that should be considered when performing this analysis, including but not limited to:

- the oracle consumer should check that the oracle has been recently updated (and call `update()` if it hasn't), to avoid reading stale data
- the basic cost estimate does not take into account liquidity effects. Namely, there is likely to be a limit to the on-chain capital that is available to arbitrageurs in the time-frame of a few blocks. For example, the attacker could manipulate the price, wait for one or more blocks to pass, before recouping some of their initial manipulation cost by performing the reversing trade. This would result in an impact on the time-weighted average price but possibly at lower cost than in a naïve estimate.
- the basic cost estimate does not take into account network congestion and network transaction cost effects. In a similar way to liquidity effects, network

congestion and costs could decrease the throughput of competing arbitrageur transactions and similarly decrease manipulation costs.
- this analysis does not consider the possibility that an attacker has the ability to mine or censor blocks. An attacker who can censor or manipulate blocks can give priority to their own transactions, and perform the manipulation at a much lower cost. As a concrete example, a miner can attempt a selfish mining-style attack where they aim to mine two or more blocks in a row before revealing them to the network: in those blocks they could include transactions leaving the price manipulated to extreme values, which could be so large as to have a meaningful impact on the time-weighted average price even if the manipulation occurs for a small number of blocks. In the worst case, the in-protocol attack cost could be limited to the LP fee levied on the size of the manipulating trade (in addition to the external costs of mining the blocks, etc.)

Swap Composability

Similarly to using a Uniswap Pair as a price oracle, smart contract developers may find it convenient to rely on a Uniswap Pair for on-chain liquidity. For example, a contract selling Cryptokitties might offer to accept payment in any token, immediately converting all proceeds from a sale to DAI via the corresponding Uniswap Pair. Similarly, a contract charging a fee for some interaction might choose to accept fees in multiple assets, provided that they can be swapped to DAI after payment.

Due to a synthesis of the "Oracle integrity" design comment above and the transaction ordering issue mentioned earlier in the report, such integrations should be designed with care, in order to prevent siphoning of funds. In particular, if the proceeds from a `swap` are not checked against a reliable, external price reference, then the contract doing the trade is effectively relying on the Pair's current price as a price oracle, in a way that is monetisable by an attacker using an atomic variant of the `swap` "sandwiching" techniques outlined above. To make matters worse, the attack against an incorrectly designed contract is practically easier to execute than the transaction sandwiching attack, since atomicity means that there is little to no execution risk and the manipulation can be funded entirely with on-chain "flash loans".

The best mitigation for this issue is for the consumer of `swap` to check the price of the resulting trade against a reliable price reference.

Timestamp and Accumulator Overflows

Calculations involving the cached timestamp (`blockTimestampLast`) and price accumulators (`price0CumulativeLast`, `price1CumulativeLast`), do not use safe math and are designed to roll over on overflow.

In the case of the timestamps, this is a performance optimisation, allowing the timestamp to be stored as a `uint32` in a single storage slot alongside the two reserves, saving two `SSTORE` operations on every call to `_update`. The timestamp will overflow once every 136 years, with the next overflow point occurring in 2106.

In the case of the accumulators, it is instead a safety measure: a revert on overflow could cause a liveness failure (a revert in `_update` would block trades, and LP entry and exit).

Although the risk of overflow is remote enough that it is not a concern for pairs of 18 decimal tokens, it can become a practical concern in pairs with tokens of mixed precision.

We can find an approximation for the time until overflow for a given pair:

$$\begin{aligned} & 2^{256} = P_{cumulative} \\ \iff\ & 2^{256} = 2^{112} \cdot \frac{Reserve_1}{Reserve_0} \cdot \Delta T \\ \iff\ & \Delta T = 2^{144} \cdot \left(\frac{Reserve_0}{Reserve_1} \right) \end{aligned}$$

Assuming that the ratio of the reserves in a given pair will be the same as the ratio of the dollar prices of one wei of each token, we can solve for a example pair consisting of a 36 decimal token and a 2 decimal token where the unit value of the 2 decimal token is 100 times that of the 36 decimal token: giving ~8 months until overflow:

$$\begin{aligned}\Delta T &= 2^{144} \cdot \left(\frac{Reserve_0}{Reserve_1}\right) \\ &= 2^{144} \cdot 10^{-36} \\ &\approx 2.23 \times 10^7 \\ &\approx 8.52\, months\end{aligned}$$

Authors of oracles that build upon the price accumulator functionality in the core should therefore take care that the their oracles do not introduce spikes or discontinuities in the reported price at the overflow point, if price accumulator overflow is a realistic possibility for the assets involved.

Expectations of Token Behaviour

A `UniswapV2Pair` contract directly interacts with its underlying pool tokens, and as such makes certain assumptions about the semantics of the `transfer` and `balanceOf` methods exposed by those tokens.

Care has been taken to make these interactions defensively, and in contrast to the V1 contracts, the V2 contracts are intended to be safe to use with:

- Potentially reentrant tokens
- Tokens that do not return from `transfer`

There are of course still ways in which a token could violate the assumptions made by the Uniswap contracts. While a full formal description of a "good" token is out of scope for this report, the audit team is aware of the following token behaviours that could cause issues with V2:

Tokens with balance changes outside of transfers

Some tokens may mutate an account's balance even when that account was not involved in a token transfer. For example, this may be used to implement (positive or negative) interest paid in-kind. An example of such a token is Ampleforth.

This may fail to have the expected economic effect on a `UniswapV2Pair`. For example, if its balance in a token were to increase outside of the usual interactions, the surplus tokens would be claimable by any account that calls `skim` (and would not result in the surplus tokens accruing to the pool, as might be expected). The same will occur if someone "airdrops" tokens on a pool by transferring tokens to it. In order to have the tokens accrue to the pool, `sync` must be called synchronously with the balance update or transfer.

`transfer` fees

Tokens that charge a fee on `transfer` break assumptions in the router, and will result in a revert when traded. Depending on their implementation they may also enable a griefing attack in the core.

Transaction fees can be implemented in two ways:

1. Deduct the fee from the amount credited to the transfer recipient
2. Transfer the full amount and deduct the fee from the sender's balance

The audit team is currently not aware of any popular tokens using the second approach, but at least PAXG is known to use the first.

Although the core is able to handle tokens of either type, both approaches are incompatible with the current implementation of the router in `uniswap-v2-periphery`. The specifics are discussed in Router: incompatible with token with fees on transfer.

Additionally, if tokens of the second type charge fees for zero balance transfers, a griefing attack will become possible where repeated calls to `skim` would allow an attacker to deplete all of the tokens in the pool as fees.

Extreme precision

The Pair stores its reserves as `uint112`, but expects the `balanceOf` method of the underlying token to return a `uint256`. The internal method (`_update`) responsible for synchronising balances to reserves therefore reverts if the balances are greater than `uint112(-1)`. This method is called during `mint`, `burn`, `swap`, and `sync`, meaning that a balance of 112 bits or more would block those methods, until `skim` is called to siphon off the excess balances. This limits the functionality of the contract for tokens where a balance in excess of 2^{112} may be realistically achieved.

For a normal 18 digit token, `uint112(-1)` represents a balance of 19,807,040,628,566,084,398,385,987,584 (1.9×10^{19} or 19 octillion), well in excess of the total supply of any mainstream token.

This limit could however become problematic should a token with a very large number of decimals be used within a `UniswapV2Pair`. The audit team is unaware of any tokens in wide use with such high precision.

Core / Periphery Seperation

The Uniswap V2 contracts introduce a separation between the core and periphery contracts, where the core contracts are responsible for supporting liquidity providers, providing time weighted price feeds and enforcing core accounting invariants. Features designed to support or protect traders are implemented with separate contracts in the periphery that call into the core.

This separation has a few benefits:

- Less code with direct access to pool tokens
- Reduced audit surface area for key invariants
- Increased amenability to the application of formal methods

However, care must be taken to correctly use either the "canonical" periphery contracts, or correctly implemented custom wrapper contracts, whenever interacting with the core. Attempting to transfer tokens to the core directly, in a separate transaction, is incorrect and could result in loss of funds.

Optimistic Swaps

The implementation of swap in UniswapV2Pair adds the ability to withdraw tokens and use them provided they are returned or paid for by the end of the transaction. At a high level, this is implemented with the following sequence of actions:

1. transfer requested output amounts to recipient
2. if the `data` argument is non-empty, call into recipient with user provided calldata
3. calculate inputs
4. check the fee adjusted constant product invariant

Notably, this sequence:

- contains a call to a user-provided account, with user-provided call-data
- transfers tokens away from the Pair before checking if it has been provided with sufficient inputs

The call into untrusted code made the application of formal methods challenging (more discussion in External call in `swap`) meaning that an exhaustive formal specification of `swap` was not produced. The audit team therefore paid particular attention to the potential for misuse in `swap`.

We are unaware of any sequence of state transitions that could lead to a violation of the fee adjusted constant product invariant as a result of a call to `swap`, for non-pathological tokens. An outline of our analysis is presented below:

Security properties

In order to extract pool tokens using swap, an attacker would have to leave the Pair in a state where the product of the Pair's reserves is less at the end of the call than it was at the start. Additionally, it should be impossible to make a successful call to `swap` that fails to compensate liquidity providers for the usage of their funds.

Specifically the following formula must be satisfied:

$$(x_1 - 0.003 \cdot x_{in}) \cdot (y_1 - 0.003 \cdot y_{in}) \geq x_0 \cdot y_0 \tag{1}$$

This ensures that:

- The constant product invariant is not violated
- Liquidity providers are paid 0.3% of the value of the inputs to the swap

Balances vs reserves

It should be noted that the above invariant is defined over the Pair's *reserves*, instead of its *balances*.

This check is similar, but not exactly equivalent. Semantically reserves can be thought of as "last recorded balances". This has the potential to be problematic as balances are arguably the thing that the `UniswapV2Pair` really cares about.

All methods that modify reserves or balances (`skim`, `sync`, `mint`, `burn`, `swap`) ensure that both are exactly matched at the end of the call. Assuming a non deflationary token, the only way to force a divergence is to transfer tokens into the Pair.

We can therefore infer a system invariant for well behaved tokens: *balances are always greater than or equal to reserves at the end of every call to* `UniswapV2Pair`.

Assuming that both of the above invariants hold we can also make some (weaker) statements about the behaviour of the constant product invariant over balances: although it is possible for the product of balances to decrease, these violations are bounded by the product of the reserves, and will in normal operation only manifest themselves through improper usage of the core (token transfer to the pool outside of an atomic sequence of transactions).

Correctness of solidity implementation

The solidity implementation of the invariant check differs from the idealised invariant above in two important ways:

1. *Arithmetic Overflow and Numeric Range*

   The solidity implementation operates on unsigned integers, and all calculations (except that for the input amounts discussed below) revert on overflow.

   This poses no concern to safety: the invariant check is correct over the domains for which it is defined, and the domain is large enough that liveness is not threatened.

2. *Calculation of Inputs*

   The calculation of input amounts does not revert on underflow. Negative inputs are instead coerced to zero. This is a design decision made to ensure that the core can handle tokens that deflate the sender's balance on transfer (perhaps as an implementation of some transfer fee). Neither the audit or Uniswap dev teams are aware of any mainstream token implemented in this manner.

   Semantically a negative input amount means that either:

  - balances were less than reserves at the start of the call
  - more than `amount{0,1}Out` tokens were transferred away from the Pair during the call

Since a negative input represents a net outflow from the contract, it is not appropriate to charge fees on them, so their coercion to zero (and the associated zero fee adjustment term) does not represent a loss for the liquidity providers.

The balances at the end of the call will reflect any unexpected shortfall or additional outflow, and the fee adjusted product of the balances at the end of the call must still exceed the product of the reserves at the start of the call. This coercion to zero does not therefore provide an avenue for a malicious caller to exploit the Pair.

Attacker controlled code execution

For the purposes of this analysis, we assume that the attacker is able to modify all blockchain state during execution of their code, except that which is protected by the mutex in the Pair.

This means that the following state transitions are unavailable during the execution of the callback:

- `skim`
- `sync`
- `mint`
- `burn`
- `swap`

Malicious code is therefore unable to:

- modify reserves
- mint or burn LP shares
- transfer pool tokens away from the Pair

It should be noted that not all methods on the Pair are protected by the reentrancy lock. Reentrant calls into the various LP share ERC20 methods (`approve`, `transfer`, `transferFrom`, `permit`) are possible. These methods do not modify state that is important for the invariant check.

The audit teams believes that these limitations are enough to ensure that an attacker cannot use reentrancy to subvert the fee adjusted invariant check: all inputs to the calculation are either fixed at the start of the call (output amounts), or protected by the mutex (reserves, balances).

Numerical Error Analysis

Locations With Possible Numerical Error

Rounding error can only occur when flooring division is used. There may also be numerical error when `sqrt` is used. With flooring division, as provided by `/` and `uqdiv`, the numerical result (interpreted as a rational number) will be less than or equal to the true ratio of the arguments (interpreted as rational numbers). With `sqrt`, the numerical result is also guaranteed to be less than or equal to the true square root.

In each case, we use this fact to determine the economic effect of any numerical error. The main properties we wish to ensure are:

- when entering the pool, numerical error should always be in favour of the liquidity providers who were already in the pool
- when leaving the pool, numerical error should always be in favour of the liquidity providers who remain in the pool
- when trading with the pool, numerical error should always be in favour of the liquidity providers
- when charging the liquidity providers with a fee, the numerical error should always be in favour of the liquidity providers

Below, we list every method where numerical error may occur:

`_update`

There is numerical error when computing the price accumulators, which has no direct economic impact on the pool itself. Consumers of accumulator data, such as oracles, should take care to ensure that the error is within acceptable bounds for their use case. Note that if a token has very low granularity, and the Pair's token balances are low, the effect of rounding down could be too high to viably use the Pair as a price oracle.

`_mintFee`

There are two square root computations, used to calculate the change in the k invariant since the last call to `_mintFee`. In principle, this can result in liquidity providers being charged a slightly higher fee than what they should be charged "in theory", if the second `sqrt` calculation happens to have a larger error than the first. However, this is not economically significant since the error is small, and most importantly does not accumulate and is bounded in time.

There is also division which rounds against the fee recipient, i.e. in favour of the existing liquidity providers.

`mint`

`sqrt` is used to set the initial LP token supply to the geometric mean of the initial supplied token amounts. Numerical error in this situation doesn't appear to have any economically relevant impact, especially in the presence of the minimum supply mitigation.

There is a division when calculating the amount of LP tokens to grant to a new contributor to the pool. The possible error here is in favour of the existing liquidity providers in the pool.

`burn`

There is division when calculating the amounts of tokens to return to a user leaving the pool. The possible error is against the user, i.e. in favour of the liquidity providers who remain in the pool.

`swap`

There are no operations that can introduce numerical error in `swap`. Instead of computing the amount to pay in exchange for the amount received (or vice-versa), the user is allowed to choose both quantities and the contract simply checks that the constant product invariant has increased by the desired amount as a result. This pushes potential rounding errors out to the caller, ensuring that numerical error cannot cause the invariant to be violated.

Proof of Correctness of `sqrt`

Below we prove that `sqrt` will terminate within 255 loop iterations, when called with any input from 0 to $2^{256}-1$ (the actual maximum number of loop iterations appears to be 135), and that return value `z` is the largest integer such that `z^2 <= y`, i.e. that `z` is the square root of `y` rounded down to the nearest integer.

Some care must be taken when handling truncating division, which is the only difference from the standard proof: let **`div(x, y)`** denote the integer result of dividing x by y without remainder. Given an integer y greater than 3, define the sequence (z_n) of positive integers by as follows:

$$z_0 = y$$
$$z_1 = \mathtt{div}(y, 2) + 1$$

and for all other n:

$$z_{n+1} = \begin{cases} \mathtt{div}(\mathtt{div}(y, z_n) + z_n, 2) & \text{if } \mathtt{div}(\mathtt{div}(y, z_n) + z_n, 2) < z_n \\ z_n & \text{otherwise} \end{cases}$$

It is obvious that this sequence stabilises at some z_N, since its tail is a sequence of monotonically decreasing positive integers. Furthermore, whenever $z_n > \sqrt{y}$, we have $z_{n+1} < z_n$, since:

$$\begin{aligned} \mathtt{div}(\mathtt{div}(y, z_n) + z_n, 2) &\le \frac{1}{2}(\mathtt{div}(y, z_n) + z_n) \\ &\le \frac{\frac{y}{z_n} + z_n}{2} \\ &< \frac{\frac{z_n^2}{z_n} + z_n}{2} = z_n \end{aligned}$$

From this it follows that the limit must satisfy $z_N \le \sqrt{y}$.

It's clear enough that this sequence models precisely the computation in `sqrt` (which we have thus proved terminates), that the limit is the return value, and N the number of iterations that the loop runs for.

It is easy to see that $N \leq y - \sqrt{y}$. We can strengthen this to a more useful bound by defining the error sequence $\epsilon_n = z_n - \sqrt{y}$, and proving that $\epsilon_n < 2^{-n}\epsilon_1$ for $n < N$ (the proof is essentially the same as in the infinite-precision setting). Thus, as $\epsilon_1 \leq 2^{255}$, we can conclude that $N < 255$ for all inputs $y < 2^{256}$.

It remains only to bound the final error $\sqrt{y} - z_N$. We do this by defining the sequence of integers r_n such that $y = z_n \cdot \mathtt{div}(y, z_n) + r_n$, so that $0 \leq r_n < z_n$ for each n, and decomposing z_N as a sum:

$$\begin{aligned} z_N &= \frac{1}{2}\left(\frac{y - r_{N-1}}{z_{N-1}} + z_{N-1}\right) \\ &= \frac{1}{2}\left(\frac{y}{z_{N-1}} + z_{N-1}\right) - \frac{z_{N-1} + r_{N-1}}{2z_{N-1}} \end{aligned}$$

The first term is strictly greater than $\sqrt{y}$, by the AM-GM inequality. The second term is strictly less than 1, due to the fact that $r_{N-1} < z_{N-1}$. This proves that $\epsilon_N < 1$, or in other words, the result `z` is the largest integer such that `z^2 <= y`. ■

Formal Verification

The Solidity compiler has introduced vulnerabilities to contracts in the past and may do so again in the future. While possible, writing bytecode by hand is arduous and prone to error. In the absence of verified compilers, formal specification of the intended behaviours of code and verification of the resulting bytecode gives strong claims of safety in areas that are not covered by non-formal code review and conventional testing.

Formal specifications for the contracts in `uniswap-v2-core` were written in the `act` specification language. These specifications are then compiled into reachability claims in the K language, and proved using the K framework prover against the bytecode produced by the build system in the `uniswap-v2-core` repository.

The `act` specifications represent a rigorous mathematical description of (almost) every possible state transition in the system. The generation of this description required a close examination of the run-time behaviour of the contract at the bytecode level. Both the specification and the process of its creation form a valuable base for higher level reasoning.

The specifications can be found in the dapp-org/k-uniswap-v2 git repo. The proof status of those specs is visible in the audit team's CI system at dapp.ci/k-uniswap.

Caveats & Assumptions

Contract level invariants

The specs as written define the functional behaviour of the invocation of a single call to a method of the contract (functional specifications).

They do not, *per se*, make claims about invariants that should hold across multiple calls (contract level invariants), such as the $xy = k$ constant product invariant (or its fee-adjusted counterpart).

The team believes that the specifications could in principle be extended to make claims about invariants at the contract level (see here for more details), but time contraints meant that such claims were not pursued as part of this engagement.

Token implementation

`UniswapV2Pair` calls into untrusted external contracts to get token balances (`balanceOf`) and to make token transfers (`transfer`). These calls could result in any behaviour, and in order to simplify the proof work, a token implementation that semantically matches the one in `uniswap-v2-core/contracts/UniswapV2ERC20.sol` has been assumed.

The proof claims are known to be valid in this context only, and in order to make formal statements about the properties of individual Pairs whose tokens may semantically deviate from this assumption, one would have to generate a new set of proofs for that token.

It should be noted that a malicious token could confiscate or burn the reserves of a Pair that included that specific token. The scope of such an attack is limited to Pairs for which the malicious token is one of the two tokens in the Pair. The loss would fall on the liquidity providers for that Pair, therefore liquidity providers should carefully audit the token implementations for each Pair that they participate in before contributing funds.

`sqrt` implementation

There are both theoretical and practical challenges with formally verifying implementations of iterative or recursive numerical algorithms. To verify the correctness of the `sqrt` implementation, one would have to check numerical error and convergence, in a fixed-precision setting, while also proving convergence and termination. A full bytecode level proof of `sqrt` would therefore be difficult and time-consuming to produce.

Due to time constraints, its implementation has not been formally verified. Instead, in proofs relying on the result of `sqrt`, a formal symbolic expression was used, meaning that the specs assert only that storage has been updated with the result of calling `sqrt`, but do not make any claims about what that result is. Note that this also assumes that the computation always terminates, assuming an adequate amount of gas was provided.

Particular care and attention has been paid to the implementation of `sqrt`, including fuzz testing, manual review, and a manual proof of correctness of the chosen algorithm in a fixed precision setting. The team discovered one subtle overflow issue. After this issue was resolved, the team is confident that the implementation is suitable for use.

External call in `swap`

If callers of `swap` provide a non empty byte array as the `data` argument, a call into a user provided address will be carried out from the Pair as part of the execution of the swap.

Calls into unknown code pose a significant challenge for the K prover. Without knowledge of the bytecode at the target of the call, the prover is unable to proceed with symbolic execution and can no longer reason about the state of the blockchain. In fact, speaking formally, one must assume that potentially *all blockchain state* could have been modified in some way by a call into unknown code. Making formal statements about blockchain state after such a call is therefore quite challenging, and to the team's knowledge, full formal verification of an Ethereum smart contract containing a call into unknown code has never been completed.

The team believes that the presence of the reentrancy mutex in `UniswapV2Pair` in combination with some custom lemmas should allow for the generation of such a proof (see Verification of external calls to unknown code), but time constraints meant that it was not pursued as part of this engagement.

The team has paid careful attention, and devoted much time to considering the potential misuse of this facility, and are unaware of any cases where its use could result in a violation of the fee adjusted constant product invariant, barring pathological token behaviours.

Reentrancy lock

The formal specifications make claims only about the state of the blockchain at the start and end of a call into the Uniswap contracts. They do not make claims about the particulars of any state transitions during a call. This means that the behaviour of the reentrancy mutex is not fully specified. The team has generated proofs of the following:

- The mutex is in the unlocked state at the start and end of all calls
- Calls into the contract will fail if the mutex is in the locked state

The specifications do not however make any claims about the transition from the unlocked to the locked state and back again within the call.

The team has paid close attention to the implementation of the mutex at the source code level, and in the resulting bytecode. Additionally, tests showing the correct functioning of the reentrancy lock have been written and observed to pass.

Exhaustiveness

Proofs of so called *ABI exhaustiveness* have been generated, meaning that the specs are known to cover all methods on the contract. Proofs of full exhaustiveness (that the specs cover the behaviour of the contract with all possible calldata values) are however not currently generated.

The team is aware of the following instances of unspecified behaviour:

- The implementation of `sqrt`
- The call into unknown code in `swap`
- Overflow of the nonces counter in `permit`
- Overflow of the `allPairs` array
- Calls to `swap` when `token0 == token1`
- Calls to `swap` when the `to` address is the address of the Pair itself
- Calls to `swap` when `blockTimestampLast` overflows
- Calls to `burn` when `totalSupply == 0`
- Calls to `burn` when the caller is the `feeTo` address
- Calls to `burn` when the `to` address is the address of the Pair, when `feeTo =/= 0` or `kLast == 0`

The `sqrt` and call into unknown code in `swap` cases are discussed in their own sections above. In the remaining cases, the decision was taken to deprioritise coverage of unlikely situations to both simplify specification and focus the teams effort on cases that were considered to be the most critical.

Soundness of software stack

A bug in `klab` or any of its dependencies could invalidate the proof results, or result in faulty specs being incorrectly accepted by the prover. The correctness of the generated proofs relies on the correctness of all of:

- The implementation of the act specification language within klab
- The K EVM semantics
- The K Framework Theorem Prover
- The Z3 SMT Prover

Soundness of lemmas

Generating the proofs required in many cases the introduction of various additional assumptions (sometimes referred to as lemmas). These assumptions are typically very narrow technical lemmas, which are infeasible to reason about with SMT solvers, but which can be easily proven mathematically. Although the team has taken care to document and justify these assumptions, an error here could potentially invalidate any proof.

The lemmas and their supporting documentation can be found in k-uniswap-v2/src/lemmas.k.md and k-uniswap-v2/src/prelude.smt2.md.

Gas

Three proof objects are generated for each method:

- `pass_rough`
- `fail_rough`
- `pass`

The `*_rough` proofs assume that the method is executed using a very large amount of gas (currently `3000000`). Once the `pass_rough` spec has been successfully proven, the trace of the symbolic execution in the proof is used to produce a symbolic expression (`G`) for the exact amount of gas used by the method (as a function of the state, calldata, etc.). This gas expression is then used to produce the `pass` proof, which proves the spec under all of the same conditions as the `pass_rough` spec, but additionally strengthens its claim to say that the amount of gas provided is `>= G`.

Due to performance constraints in the `K` prover, a proof that the method will fail if called with a gas value `< G` is currently not generated. However, the team believes that this claim should in principle be provable.

Future Work

The audit team believes that the existing formal specifications could be extended or built upon in several valuable directions, and encourages the Uniswap team to investigate the following in the future:

Contract level invariants

The audit team believes that the act specs generated as part of this engagement could be extended to prove that core accounting invariants are maintained for every possible combination of calls into a V2 Pair.

The team believes that two potential approaches are viable:

1. Compilation of the act specs into some high level proof language (e.g. Coq,

Agda)

2. An inductive proof implemented using post-conditions in the generated K specs

Verification of external calls to unknown code

The team believes that the addition of lemmas resetting all EVM state that could potentially have been affected by the call (essentially all state not guarded by the reentrancy mutex) to an abstract value after the conclusion of the call should allow for a successful proof of correctness of the relevant code paths in `swap`.

Verification of `sqrt` bytecode

The team believes that it should be possible to prove a loop invariant by coïnduction showing the EVM implementation of `sqrt` agrees with the model used in the manual proof of convergence.

Appendix A. Bug Classifications

Severity	
informational	The issue does not have direct implications for functionality, but could be relevant for understanding.
low	The issue has no security implications, but could affect some behaviour in an unexpected way.
medium	The issue affects some functionality, but does not result in economically significant loss of user funds.
high	The issue can cause loss of user funds.
Likelihood	
low	The system is unlikely to be in a state where the bug would occur or could be made to occur by any party.
medium	It is fairly likely that the issue could occur or be made to occur by some party.

Severity	
high	It is very likely that the issue could occur or could be exploited by some parties.

Footnotes:

[1] Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchange - Philip Daian

[2] Taking undercollateralised loans for fun and profit - samczsun

[3] An Analysis of Uniswap Markets - Angeris, Kao, Chiang, Noyes, Chitra