CERTORA

Formal Verification of Compound's Open-Oracle With Uniswap Anchor

1. Summary

This document describes the specification and verification using Certora Prover of Compound's Open-Oracle framework with a Uniswap V2 based price anchoring. The work was undertaken from June 28th through July 19th, 2020. The latest commit that was reviewed and run through the Certora Prover was c77dab0e3fa92ce1872227833b740c445046844b.

The scope was the Open Oracle framework with the Uniswap based anchor, and the lagging window mechanism. The main contract considered was UniswapAnchoredView.

The Certora Prover proved the implementation of the Open Oracle correct with respect to the formal rules written. During the verification process, the Certora Prover discovered a number of bugs in the code listed in the table below in section 1.1. All the high severity issues were promptly corrected, and the fixes were verified to satisfy the specifications. Section 2 formally defines high level specifications of the protocol.

1.1. List of Main Issues Discovered

Issue	Affected code	Description	Severity	Mitigation
Incorrect cTokens prices' calculation	UniswapAnchoredView	The system returns the wrong price of cTokens with FIXED_ETH priceSource due to erroneous division by baseUnit**2.	High	Replaced the division by baseUnit with a division by ethBaseUnit in *priceInternal*.
Anchor TWAP computed over too small time intervals during initialization	UniswapAnchoredView	During the first *anchorPeriod* since initialization, the anchor TWAP is computed using time intervals smaller than *anchorPeriod*.	Medium	Assuming that anchor TWAP will not be computed during the first *anchorPeriod*.

Invalid parameter check in *postPrices*	UniswapAnchoredView	Reverting due to index out of bounds. Skipping posting of a symbol's price whose corresponding message is not from the reporter.	Medium	Removed redundant check.
Error-prone maintainability of UniswapConfig	UniswapConfig	***getTokenConfig(uint)*** function could return an undefined value. This may occur if the contact is changed/upgraded such that ***numTokens*** and ***maxTokens*** fields are increased beyond 30 but ***getTokenConfig(uint)*** remains unchanged.	Low	Assuming correct initialization and maintenance / upgrade.
Non-existing TokenConfigs have FIXED_ETH priceSource	UniswapConfig and UniswapAnchoredView	The ***priceSource*** of a non-existing ***TokenConfig*** is FIXED_ETH (0). Such a config is treated as a valid FIXED_ETH one in ***priceInternal(TokenConfig config)***. This leads to a division by zero in ***getUnderlyingPrice***.	Low	Assuming correct initialization.
Correctness relies on Uniswap cumulative price to wrap around at most one cycle	UniswapAnchoredView	If there are two wrap arounds, the anchor TWAP would be calculated incorrectly.	Low	Cumulative prices are expected to overflow. Comparing prices that are at most 30 minutes apart to ensure they are at most one wrap around separate. This time interval can be guaranteed by setting a not too big ***anchorPeriod***.
Correctness	UniswapAnchoredView	If uniswap cumulative	Low	Uniswap is assumed to

relies on increasing Uniswap cumulative prices		price is not strictly increasing during a price observation update cycle, the anchor TWAP calculation is incorrect.		be correct. A single overflow in a cycle is permitted.
A once-invalidated reporter address should not be re-used with the same Oracle implementation	UniswapAnchoredView	The rotate message can be replayed.	Low	Make sure not to re-deploy the oracle contract with a reporter address previously invalidated.
Big price movements are possible (1)	UniswapAnchoredView	When the reporter is invalidated, the Uniswap price serving as anchor is taken as the asset's price.	Low	After the reporter is invalidated, re-deploy with a new reporter promptly.
Big price movements are possible (2)	UniswapAnchoredView	If Uniswap anchor is farther from the threshold compared to the reporter, and the price is guarded, then a post where the reporter aligns with the anchor will lead to a price jump.	Low	This is a known property of the system and no mitigation is needed.
Timestamp vs. Block number	UniswapAnchoredView	To guarantee anchor price updates are at least ***anchorPeriod*** apart, block timestamps are used. The correctness of this measurement is vulnerable to skews in the timestamps reported by miners, as if more time elapsed than in reality.	Low	Inherent trust in the Ethereum network to avoid timestamps skews, therefore the risk is low if ***anchorPeriod*** is set to, say, 30 minutes.

1.2. Technology Overview

The Certora Prover is based on well-studied techniques from the formal verification community. Specifications define a set of rules that call into the contract under analysis and make various assertions about their behavior. These rules, together with the contract under analysis, are compiled to a logical formula called a verification condition, which is then proved or disproved by the solver Z3. If the rule is disproved, the solver also provides a concrete test case demonstrating the violation.

The rules of the specification play a crucial role in the analysis. Without good rules, only very shallow properties can be checked (e.g. that no assertions in the contract itself are violated). To make effective use of Certora Prover, users must write rules that describe the high-level properties they wish to check of their contract. Certora Prover cannot make any guarantees about cases that fall outside the scope of the rules provided to it as input. Thus, in order to understand the results of this analysis, one must carefully understand the specification's rules.

1.3. Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

2. High Level Specification

Functions used in the verification process:

Getters:

price(symbol string) returns (uint)

Returns the price of the specified symbol hash.

newObservation(symbol string) returns (Observation)

Returns the new Uniswap cumulative anchor price observed by the system.

oldObservation(symbol string) returns (Observation)

Returns the old Uniswap cumulative anchor price observed by the system.

anchorPeriod() returns (uint)

Returns the value of ***anchorPeriod*** that is used to update the cumulative price observations.

eth() returns (string)

Returns the Ether symbol.

reporterInvalidated() returns (bool)

Returns true if the reporter has invalidated itself.

Methods:

For readability, we list the functions' signatures used in the properties below.

put(signer address, symbol string, timestamp uint64, value uint64)

Writes price from signer into the Open Oracle Price Data contract.

fetchAnchorPrice(symbol string, currTimestamp uint) returns (uint)

May update new and old observation info:

- timestamp
- accumulate value

And returns the new anchor price (TWAP), average for the delta time between currTimestamp and the old observation's timestamp.

pokeWindowValues(symbol string, currTimestamp uint) returns (uint, uint, uint)

Returns cumulative price for current block timestamp (currTimestamp), the old observation's cumulative price and the old observation's timestamp.

postPrice(symbol string, ethPrice uint, currTimestamp uint) returns (bool) {

```
    require(block.timestamp == currTimestamp);
    postPriceInternal(symbol, ethPrice);
    return !lastReverted;
}
```

Invokes *postPriceInternal(string, uint)* to compute the price for *symbol* based on the current state of the Open Oracle Price Data contract and on a new fetch of the anchor price, which relies on *ethPrice*. Verifies conditions for the new price and updates the stored price if all conditions are met. Returns false when failed (i.e., if there was a revert); otherwise returns true.

Properties:

✔️ Consistent observations' timestamps

```
consistent_timestamps(sym) ≡
        oldObservation(sym).timestamp = newObservation(sym).timestamp ∨
        newObservation(sym).timestamp ≥ oldObservation(sym).timestamp + anchorPeriod()
∀sym:symbol. consistent_timestamp(sym)
```

❌ Poke window delta time - general case

On successful computation of PokeWindow, time used to compute anchor average price is at least anchorPeriod. Here we use Hoare triples of the form `{p} C {q}`, which means that if program C executes starting in any state satisfying `p`, then it will end in a state satisfying `q`.

Assuming the system is in a state fulfilling the observation timestamp invariant

```
{ consistent_timestamps(sym) }
        (cum, oldAcc, oldTimestamp) = pokeWindowValues(sym, block.timestamp)
{ block.timestamp ≥ oldTimestamp + anchorPeriod() }
```

This is violated with the system is in initialization state and therefore we prove two refined properties below:

✔ Poke window delta time - initialization end

Assuming the system is in initialization state, after anchorPeriod the system reaches the regular state.

```
{ newObservation(sym).timestamp = oldObservation(sym).timestamp ⋀
  block.timestamp ≥ oldObservation(sym).timestamp + anchorPeriod() }
      (cum, oldAcc, oldTimestamp) = pokeWindowValues(sym, block.timestamp)
{ newObservation(sym).timestamp ≥ oldObservation(sym).timestamp + anchorPeriod() }
```

✔ Poke window delta time - after initialization

Assuming the system is in a state after initialization, poke window is as expected.

```
{ newObservation(sym).timestamp ≥ oldObservation(sym).timestamp + anchorPeriod() }
      (cum, oldAcc, oldTimestamp) = pokeWindowValues(sym, block.timestamp)
{ block.timestamp ≥ oldTimestamp + anchorPeriod() }
```

✔ Price Idempotent

With the same open oracle price data, and once a symbol's price is posted, its price should remain unchanged. This does not hold when the reporter is invalidated, as the price changes to the anchor price on every post.

```
{ p1 ≠ p0 ⋀ ¬reporterInvalidated() }
      p0 = price(s);
      postPrice(s, e, t1);
      p1 = price(s);
      postPrice(s, e, t2);
      p2 = price(s);
{ p1 = p2 }
```

✔ No change to other symbols

When posting a price for a symbol, the prices of other symbols do not change. The only exception is the Ether (eth) price, which influences Ether denominated symbols' prices.

```
{ sym ≠ sym' ⋀ p = price(sym') ⋀ sym ≠ eth() }
      put(source, sym, timestamp, value); postPrice(sym, price(eth()), timestamp')
{ p = price(sym') }
```

❔ Commutativity of Put messages

The order of put messages does not affect the price of a symbol.

This is specified in terms of code equivalence with respect to getter op, denoted by $P_1 \sim P_2$. Conditions are expressed with a subscript to indicate as to which run they refer to. These properties are called relational properties. The syntax {Pre} $P_1 \sim P_2$ {Post} is a generalization of Hoare rules. Pre is a requirement on the states before P1 and P2 and {Post} describes the states after their executions. Notice that P1 and P2 operate in different states.

```
{ t ≠ t' }
put(a, sym, t, v); put(a', sym, t', v'); postPrice(sym, e, t'')
∼
put(a', sym, t', v'); put(a, sym, t, v); postPrice(sym, e, t'')
{price(sym)₁ = price(sym)₂ }
```

Here, $price_1$ is the price after the first transaction and $price_2$ is the price after the second transaction.

Currently, this rule times out.

❔ Revert characteristics of PostPrice

The PostPrice function does not revert under the following necessary assumptions:

- Symbol is valid, namely it has a config struct.
- The symbol config struct is initialized properly.
- The new and old observation timestamps are consistent.
- The current block timestamp is not before the new observation timestamp.
- During the first anchor period, the current block timestamp is greater than the old observation timestamp.
- The anchor period is non-zero.
- No multiplication overflows occur.
- msg.value == 0.
- PriceData contract has a getPrice function.

```
{ consistent_timestamp(sym) ∧ t ≥ newObservation(sym).timestamp ∧ anchorPeriod() > 0 }
        ok = postPrice(sym, e, t)
{ ok }
```