Formal Verification Report of Rolla

Summary

This document describes the specification and verification of Rolla Protocol using the Certora Prover. The work was undertaken from June 7 - 24, 2021. The rules are running as part of CI.

The scope of our verification was the Controller, CollateralToken and FundsCalculator contracts.

The Certora Prover proved the implementation of the Rolla Protocol is correct with respect to the formal rules written by the Rolla team and the Certora teams. During the verification process, the Certora Prover discovered bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of Rolla Protocol.

All the rules are publically available in a public github.

Certora Prover run results: CollateralToken FundsCalculator Controller

List of Main Issues Discovered

Severity: Critical

Issue:	Lose of all system assets
Rules Broken:	Valid QToken

Issue:	Lose of all system assets
Description:	By crafting a malicious qToken and exercising options that were not minted in the system, one can drain all collateral of the system
Mitigation/Fix:	Allow only valid QTokens

Severity: **Medium**

Issue:	Denial of service on neutralizing position
Rules Broken:	-
Description:	Due to wrong computation, when trying to neutralize the maximum possible position, the call will revert
Mitigation/Fix:	Fixed

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 the Certora Prover does not check any cases not covered by the specification.

We hope that this information is useful, but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

✔️ indicates the rule is formally verified on the latest reviewed commit. We write ✔️* when the rule was verified on a simplified version of the code (or under some assumptions).

`![](https://i.imgur.com/rDhiM7e.png =20x20)` indicates the rule was violated under one of the tested versions of the code.

✍️ indicates the rule is not yet formally specified.

🔁 indicates the rule is postponed (<due to other issues, low priority>) .

We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.

The syntax {p} (C1 ～ C2) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C1 and C2, and {q} describes the states after their executions. Notice that C1 and C2 result in different states. As a special case, C1～op C2, where op is a getter, indicating that C1 and C2 result in states with the same value for op.

Verification of CollateralToken

CollateralToken is a ERC1155 Token representing a Rolla user's short positions. Can be used by owners to claim their collateral.

Properties

1. TotalSupply is the sum of balances ✔️

For each collateralTokenId, the total supply is the sum of balances of this token id

```
totalSupplies[collateralTokenId] = ∑ address x.  _balances[collateralTokenId][x]
```

2. Uniqueness collateralTokenIds ✔️

Each entry in collateralTokenIds is unique

```
collateralTokenIds[i] = collateralTokenIds[j] ⇒ i = j
```

3. Integrity of CollateralTokenInfo ✔️

Creating a new pair of QTokens creates the collateralTokenInfo

```
{ }
    collateralTokenInfoId = createCollateralToken(qTokenAddress,
qTokenAsCollateral);
{ (qTokenAddress, qTokenAsCollateral) =
getCollateralTokenInfo(collateralTokenInfoId) }
```

4. Integrity of CollateralTokenIdToInfo ✔️

Each collateralToken has an entry in the collateralTokenInfo

```
{ i = collateralTokenIds.length }
    collateralTokenInfoId = createCollateralToken(qToken, qTokenAsCollateral)
{ idToInfo[collateralTokenInfoId].qTokenAddress ≠ 0 ∧ collateralTokenIds[i] = key
}
```

5. Integrity of minting ✔️

On minting, the balance is updated by the amount minted

```
{ b = balanceOf(_recipient, collateralTokenInfoId) }
        mintCollateralToken(e, recipient, collateralTokenInfoId, amount);
{ balanceOf(recipient, collateralTokenInfoId) = amount + b}
```

6. Integrity of token supply on Minting ✔️

Once minting, the collateralToken has an entry in the tokenSupplies array

```
{ amount > 0 }
    collateralTokenInfoId = createCollateralToken(qToken, qTokenAsCollateral);
        mintCollateralToken(e, _recipient, collateralTokenInfoId, _amount);
{ tokenSupplies[collateralTokenInfoId] ≠ 0 }
```

7. Burn after mint ✔️

Once minting, the collateralToken can be burned

```
{before = balanceOf(user, collateralTokenInfoId)  }
    mintCollateralToken(user, collateralTokenInfoId, _amount);
        burnCollateralToken(user, collateralTokenInfoId, _amount)
{ balanceOf(user, collateralTokenInfoId) = before }
```

Verification of FundsCalculator

FundsCalculator library contains the mathematical functions for computing collateral requirements, payout, claimable amount.

9. Spreads require less collateral than minting options ✔️

Minting spreads (for both calls and puts) require same or less collateral than minting options.

```
getOptionCollateralRequirement(qTokenToMintStrikePrice,
qTokenForCollateralStrikePrice, optionsAmount)  ≤
getOptionCollateralRequirement(qTokenToMintStrikePrice, 0, optionsAmount)
```

10. Put collateral increases with decrease in collateral strike price ✔️

Put spreads require same or more collateral as the collateral option token strike price decreases.

```
collateralStrikePrice1 > collateralStrikePrice2 ⇒
getPutCollateralRequirement(mintStrikePrice, collateralStrikePrice2) ≥
getPutCollateralRequirement(mintStrikePrice, collateralStrikePrice1)
```

11. Put collateral decreases with decreases in mint strike price ✔️

Put spreads require same or less collateral as the mint option token strike price decreases.

```
mintStrikePrice1 > mintStrikePrice2 ⇒
getPutCollateralRequirement(mintStrikePrice2, collateralStrikePrice) ≤
getPutCollateralRequirement(mintStrikePrice1, collateralStrikePrice)
```

12. Call spread collateral increases with increase in collateral strike price ✔️

Call spreads require same or more collateral requirement as the collateral option token strike price increases.

```
collateralStrikePrice1 > 0 ∧ collateralStrikePrice2 > 0 ∧
collateralStrikePrice1 < collateralStrikePrice2 ⇒
getCallCollateralRequirement(mintStrikePrice, collaterStrikePrice2) ≥
getCallCollateralRequirement(mintStrikePrice, collateralStrikePrice1)
```

13. Call spread collateral decreases with increases in mint strike price ✔️

Call spreads require same or less collateral requirement as the mint option strike price increases.

```
mintStrikePrice1 < mintStrikePrice2 ⇒
getCallCollateralRequirement(mintStrikePrice2, collateralStrikePrice) ≤
```

```
getCallCollateralRequirement(mintStrikePrice1, collateralStrikePrice)
```

14. Positive Put Payout ✔️

Payout for Puts is positive if and only if strike price is greater than expiry price and options amount is greater than 0.

```
getPayoutForPut(strikePrice, expiryPrice, amount) > 0 ⇔
(strikePrice > expiryPrice) ∧ (optionsAmount > 0)
```

15. Positive Call Payout ✔️

If payout for a Call is positive, then expiry price must be greater than strike price and options amount must be positive.

```
getPayoutForCall(strikePrice, expiryPrice, amount) > 0 ⇒
(expiryPrice > strikePrice) ∧ (optionsAmount > 0)
```

16. Zero Payout amount ✔️

If expiry price is equal to strike price of an option or options amount is zero, the payout amount then is also zero.

```
(expiryPrice = strikePrice) v (amount = 0) ⇒ getPayoutAmount(_isCall,
strikePrice, expiryPrice, amount) = 0
```

17. Additive Call Payout amount ✔️

Payout amount for Call options follows additivity, i.e. PayoutAmount for amount1 + amount2 is equal to the sum of PayoutAmounts for amount1 and amount2.

```
getPayoutforCall(strikePrice, expiryPrice, amount1) +
getPayoutForCall(strikePrice, expiryPrice, amount2) =
getPayoutForCall(strikePrice, expiryPrice, amount1 + amount2)
```

18. Additive Put Payout amount ✔️

Payout amount for Put options follows additivity, i.e. PayoutAmount for amount1 + amount2 is equal to the sum of PayoutAmounts for amount1 and amount2.

```
getPayoutforPut(strikePrice, expiryPrice, amount1) + getPayoutforPut(strikePrice, expiryPrice, amount2) =
getPayoutforPut(strikePrice, expiryPrice, amount1 + amount2)
```

Verification of Controller

The main contract for minting, exercising and claiming options. For ease of reading, in all rules when referring to a collateralID and QToken, we assume they are correlated

General properties

19. Balance VS supply ✔️

totalSupply always greater than balance of user

```
qTokenA.totalSupply() ≥ qTokenA.balanceOf(e.msg.sender)
```

20. Valid QToken `![](https://i.imgur.com/rDhiM7e.png =20x20)`

Only valid QToken can be used in functions that change the QToken's totalSupply

```
{ t = qToken.totalSupply() }
        op
{ qToken.totalSupply() ≠ t ⇒ qToken.isValid() }
```

21. Solvency ✔️

The user can not gain excess assets or lose assets. User's total asset is computed as the balanceOf in the asset, the value of his qTokens and the value of his collateralTokens

```
{ before = asset.balanceOf(u) +  exercisePayout(qTokenA.balanceOf(u)) +
claimableCollateral(collateralTokenId, collateralToken.balanceOf(u,
collateralTokenId)) }
          op
```

```
{ asset.balanceOf(u) +  exercisePayout(qTokenA.balanceOf(u)) +
claimableCollateral(collateralTokenId, collateralToken.balanceOf(u,
collateralTokenId)) = before }
```

where op is not minstSpread

22. Integrity Of Totals ✔️

User owns all the qTokens if and only if user owns all the collateral tokens

```
{ qToken.balanceOf(u) = qToken.totalSupply()  ⇔  Collateral.balanceOf(u) =
Collateral.totalSupply() }
                op
{ qToken.balanceOf(u) = qToken.totalSupply()  ⇔  Collateral.balanceOf(u) =
Collateral.totalSupply() }
```

23. Same Asset Token ✔️

Functions getExercisePayout, getCollateralRequirement, calculateClaimableCollateral return the same ERC20token for the same qtoken/collateralTokenID

```
(_, payoutToken, _ ) = getExercisePayout(qToken,y) ∧
(_, asset, _) = calculateClaimableCollateral(cID,x,u) ∧
(collateral, _) = getCollateralRequirement(cID,x,u)

 ⇒ payoutToken = asset ∧ asset = collateral
```

Neutralize Options Properties

24. Ratio after neutralize ✔️

increase in qTokens equals decrease in collateral tokens

```
{ mintBefore = mint.totalSupply() ∧
  CollBefore = collateralToken.getTokenSupplies(cId)}
     neutralizePosition(cId, amount)
{ mint.totalSupply() - mintBefore = collateralToken.getTokenSupplies(cId) -
CollBefore }
```

25. Neutralize burn correctness ✔️

Neutralizing options must always burn the amount passed into the function

```
{ mintBefore = qTokenA.totalSupply() ∧ forCollBefore = qTokenB.totalSupply() ∧
amount > 0 }
        neutralizePosition(cId, amount)
{ qTokenA.totalSupply() = mintBefore - amount XOR qTokenB.totalSupply() =
forCollBefore - amount }
```

Minting Options Rules

26. Integrity of Mint Options ✔️

Changes to balanceOf and totalSupply is as expected

```
{ b = qToken.balanceOf(to)  ∧ t = qToken.totalSupply() ∧
  bCid = collateralToken.balanceOf(to,tokenId) ∧
tCid = collateralToken.tokenSupplies(tokenId) }
        mintOptionsPosition(to,qToken,amount)
{ qToken.balanceOf(to) =  b + amount ∧
  qToken.totalSupply() = t + amount ∧
  collateralToken.balanceOf(to,tokenId) = bCid + amount ∧
  collateralToken.tokenSupplies(tokenId) = tCid + amount }
```

27. Mint options correctness ✔️

Increase in user's balance equals decrease in the contract's balance in the respective token

```
{ beforeUser = asset.balanceOf(u) ∧ beforeContract = asset.balanceOf(c) }
    mintOptionsPosition(u, qToken, amount)
{ beforeUser - asset.balanceOf(u) = asset.balanceOf(c)  - beforeContract }
```

28. Additive minting ✔️

Minting options in steps requires the same or more collateral than minting all in one go

```
mintOptionsPosition(u, qToken, x);mintOptionsPosition(u, qToken, y)
~
mintOptionsPosition(u, qToken, x + y)
```

with respect to collateralToken.balanceOf(u, cId)

Exercising Options Rules

29. Exercise burn correctness ✔️

Exercising options must always burn the amount passed into the function

```
{ before = qTokenA.totalSupply() }
    exercise(qToken, amount)
{ qTokenA.totalSupply() = before - amount }
```

30. Balances after exercise ✔️

increase in user's balance equals decrease in the contract's balance in the respective token

```
{ beforeUser = asset.balanceOf(u) ∧ beforeContract = asset.balanceOf(system) }
    exercise(qToken, amount);
{ asset.balanceOf(u) - beforeUser = beforeContract - asset.balanceOf(system) }
```

31. Exercise only after expiry ✔️

exercise should fail if called before expiry

Minting Spread Options Rules

32. MintSpread balances correctness ✔️

Minting spreads must burn the qTokens provided as collateral, mint the desired qTokens and also mint collateral tokens representing the spread

Claiming Collateral Rules

33. Additive claim ✔️

Claim collateral of x and then of y produces same result as claim collateral of (x+y)

```
claimCollateral(cId, x); claimCollateral(cId, y)
~
claimCollateral(cID, x+y)
```

with respect to collateralToken.balanceOf(e.msg.sender, cId)

34. Zero collateral zero claim ✔️

Claiming collateral having zero collateral result in zero claimed

```
{ b = collateralToken.balanceOf(u, collateralTokenId) = 0 ∧ c =
getTokenBalanceOf(asset, system) }
    claimCollateral(collateralTokenId, amount);
{ b = 0 ⇒ getTokenBalanceOf(asset, system) = c }
```